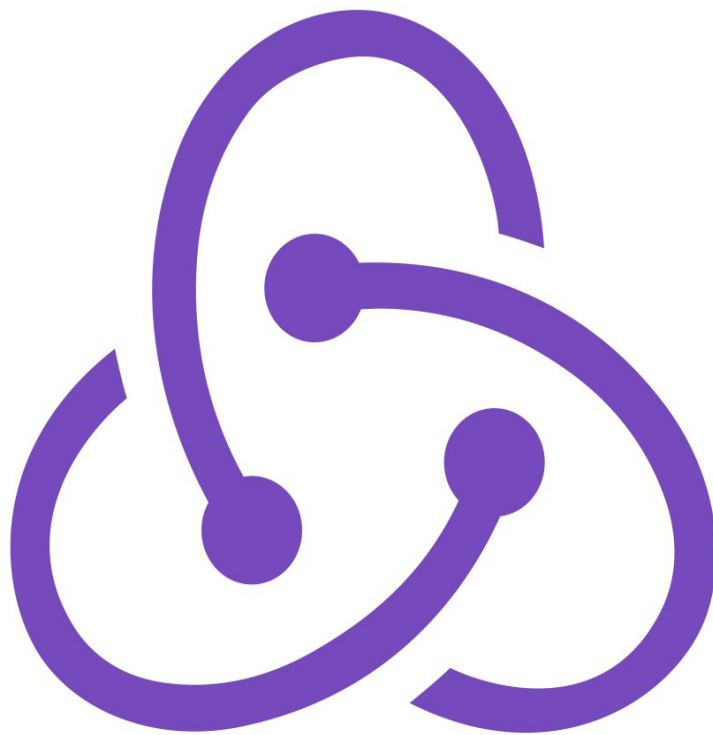


Redux v4.0.5

Documentation



Redux

**Redux Official
Documentation**

Table of Contents

Introduction	1.1
Read Me	1.2
Introduction	1.3
Motivation	1.3.1
Core Concepts	1.3.2
Three Principles	1.3.3
Prior Art	1.3.4
Learning Resources	1.3.5
Ecosystem	1.3.6
Examples	1.3.7
Basics	1.4
Actions	1.4.1
Reducers	1.4.2
Store	1.4.3
Data Flow	1.4.4
Usage with React	1.4.5
Example: Todo List	1.4.6
Advanced	1.5
Async Actions	1.5.1
Async Flow	1.5.2
Middleware	1.5.3
Usage with React Router	1.5.4
Example: Reddit API	1.5.5
Next Steps	1.5.6
Recipes	1.6
Configuring Your Store	1.6.1
Migrating to Redux	1.6.2
Using Object Spread Operator	1.6.3
Reducing Boilerplate	1.6.4
Server Rendering	1.6.5
Writing Tests	1.6.6
Computing Derived Data	1.6.7
Implementing Undo History	1.6.8
Isolating Subapps	1.6.9
Structuring Reducers	1.6.10

Prerequisite Concepts	1.6.10.1
Basic Reducer Structure	1.6.10.2
Splitting Reducer Logic	1.6.10.3
Refactoring Reducers Example	1.6.10.4
Using combineReducers	1.6.10.5
Beyond combineReducers	1.6.10.6
Normalizing State Shape	1.6.10.7
Updating Normalized Data	1.6.10.8
Reusing Reducer Logic	1.6.10.9
Immutable Update Patterns	1.6.10.10
Initializing State	1.6.10.11
Using Immutable.JS with Redux	1.6.11
FAQ	1.7
General	1.7.1
Reducers	1.7.2
Organizing State	1.7.3
Store Setup	1.7.4
Actions	1.7.5
Immutable Data	1.7.6
Code Structure	1.7.7
Performance	1.7.8
Design Decisions	1.7.9
React Redux	1.7.10
Miscellaneous	1.7.11
Troubleshooting	1.8
Glossary	1.9
API Reference	1.10
createStore	1.10.1
Store	1.10.2
combineReducers	1.10.3
applyMiddleware	1.10.4
bindActionCreators	1.10.5
compose	1.10.6
Change Log	1.11
Patrons	1.12
Feedback	1.13

Redux **v4.0.5** Documentation



Redux

**Redux Official
Documentation**

Introduction

- [Motivation](#)
- [Core Concepts](#)
- [Three Principles](#)
- [Prior Art](#)
- [Learning Resources](#)
- [Ecosystem](#)
- [Examples](#)

Motivation

As the requirements for 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. If a model can update another model, then a view can update a model, which updates another model, and this, in turn, might cause another view to update. At some point, you no longer understand what happens in your app as you have **lost control over the when, why, and how of its state**. When a system is opaque and non-deterministic, it's hard to reproduce bugs or add new features.

As if this weren't bad enough, consider the **new requirements becoming common in front-end product development**. As developers, we are expected to handle optimistic updates, server-side rendering, fetching data before performing route transitions, and so on. We find ourselves trying to manage a complexity that we have never had to deal with before, and we inevitably ask the question: [is it time to give up?](#) The answer is *no*.

This complexity is difficult to handle as **we're mixing two concepts** that are very hard for the human mind to reason about: **mutation and asynchronicity**. I call them [Mentos and Coke](#). Both can be great in separation, but together they create a mess. Libraries like [React](#) attempt to solve this problem in the view layer by removing both asynchrony and direct DOM manipulation. However, managing the [state](#) of your data is left up to you. This is where Redux enters.

Following in the steps of [Flux](#), [CQRS](#), and [Event Sourcing](#), **Redux attempts to make state mutations predictable** by imposing certain restrictions on how and when updates can happen. These restrictions are reflected in the [three principles](#) of Redux.

Core Concepts

Imagine your app's **state** is described as a plain object. For example, the **state** of a todo app might look like this:

```
{
  todos: [{
    text: 'Eat food',
    completed: true
  }, {
    text: 'Exercise',
    completed: false
  }],
  visibilityFilter: 'SHOW_COMPLETED'
}
```

This object is like a “model” except that there are no setters. This is so that different parts of the code can’t change the **state** arbitrarily, causing hard-to-reproduce bugs.

To change something in the **state**, you need to dispatch an **action**. An **action** is a plain JavaScript object (notice how we don’t introduce any magic?) that describes what happened. Here are a few example actions:

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Enforcing that every change is described as an **action** lets us have a clear understanding of what’s going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie **state** and actions together, we write a function called a **reducer**. Again, nothing magical about it—it’s just a function that takes **state** and **action** as arguments, and returns the next **state** of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the **state**:

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  if (action.type === 'SET_VISIBILITY_FILTER') {
    return action.filter
  } else {
    return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map((todo, index) =>
        action.index === index
          ? { text: todo.text, completed: !todo.completed }
          : todo
      )
    default:
      return state
  }
}
```


And we write another **reducer** that manages the complete **state** of our app by calling those two reducers for the corresponding **state** keys:

```
function todoApp(state = {}, action) {  
  return {  
    todos: todos(state.todos, action),  
    visibilityFilter: visibilityFilter(state.visibilityFilter, action)  
  }  
}
```

This is basically the whole idea of Redux. Note that we haven't used any Redux APIs. It comes with a few utilities to facilitate this pattern, but the main idea is that you describe how your **state** is updated over time in response to **action** objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic.

Three Principles

Redux can be described in three fundamental principles:

Single source of truth

The **state** of your whole application is stored in an object tree within a single **store**.

This makes it easy to create universal apps, as the **state** from your server can be serialized and hydrated into the client with no extra coding effort. A single **state** tree also makes it easier to debug or inspect an application; it also enables you to persist your app's **state** in development, for a faster development cycle. Some functionality which has been traditionally difficult to implement - Undo/Redo, for example - can suddenly become trivial to implement, if all of your **state** is stored in a single tree.

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

State is read-only

The only way to change the **state** is to emit an **action**, an object describing what happened.

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. As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

Changes are made with pure functions

To specify how the **state** tree is transformed by actions, you write pure **reducers**.

Reducers are just pure functions that take the previous [state](#) and an [action](#), and return the next [state](#). Remember to return new [state](#) objects, instead of mutating the previous [state](#). You can start with a single [reducer](#), and as your app grows, split it off into smaller reducers that manage specific parts of the [state](#) tree. Because reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}

import { combineReducers, createStore } from 'redux'
const reducer = combineReducers({ visibilityFilter, todos })
const store = createStore(reducer)
```

That's it! Now you know what Redux is all about.

Prior Art

Redux has a mixed heritage. It is similar to some patterns and technologies, but is also different from them in important ways. We'll explore some of the similarities and the differences below.

Flux

Redux was inspired by several important qualities of [Flux](#). Like Flux, Redux prescribes that you concentrate your model update logic in a certain layer of your application ("stores" in Flux, "reducers" in Redux). Instead of letting the application code directly mutate the data, both tell you to describe every mutation as a plain object called an "action".

Unlike Flux, **Redux does not have the concept of a Dispatcher**. This is because it relies on pure functions instead of event emitters, and pure functions are easy to compose and don't need an additional entity managing them. Depending on how you view Flux, you may see this as either a deviation or an implementation detail. Flux has often been [described as](#) `(state, action) => state`. In this sense, Redux is true to the Flux architecture, but makes it simpler thanks to pure functions.

Another important difference from Flux is that **Redux assumes you never mutate your data**. You can use plain objects and arrays for your [state](#) just fine, but mutating them inside the reducers is strongly discouraged. You should always return a new object, which is easy with the [object spread operator proposal](#), or with a library like [Immutable](#).

While it is technically *possible* to [write impure reducers](#) that mutate the data for performance corner cases, we actively discourage you from doing this. Development features like time travel, record/replay, or hot reloading will break. Moreover it doesn't seem like immutability poses performance problems in most real apps, because, as [Om](#) demonstrates, even if you lose out on object allocation, you still win by avoiding expensive re-renders and re-calculations, as you know exactly what changed thanks to [reducer](#) purity.

For what it's worth, Flux's creators [approve](#) of [Redux](#).

Elm

[Elm](#) is a functional programming language inspired by Haskell and created by [Evan Czaplicki](#). It enforces a ["model view update" architecture](#), where the update has the following signature: `(action, state) => state`. Elm "updaters" serve the same purpose as reducers in Redux.

Unlike Redux, Elm is a language, so it is able to benefit from many things like enforced purity, static typing, out of the box immutability, and pattern matching (using the `case` expression). Even if you don't plan to use Elm, you should read about the Elm architecture, and play with it. There is an interesting [JavaScript library playground implementing similar ideas](#). We should look there for inspiration on Redux! One way that we can get closer to the static typing of Elm is by [using a gradual typing solution like Flow](#).

Immutable

[Immutable](#) is a JavaScript library implementing persistent data structures. It is performant and has an idiomatic JavaScript API.

Immutable and most similar libraries are orthogonal to Redux. Feel free to use them together!

Redux doesn't care *how* you **store the state**—it can be a plain object, an Immutable object, or anything else. You'll probably want a (de)serialization mechanism for writing universal apps and hydrating their **state** from the server, but other than that, you can use any data storage library *as long as it supports immutability*. For example, it doesn't make sense to use Backbone for Redux **state**, because Backbone models are mutable.

Note that, even if your immutable library supports cursors, you shouldn't use them in a Redux app. The whole **state** tree should be considered read-only, and you should use Redux for updating the **state**, and subscribing to the updates. Therefore writing via cursor doesn't make sense for Redux. **If your only use case for cursors is decoupling the **state** tree from the UI tree and gradually refining the cursors, you should look at selectors instead.** Selectors are composable getter functions. See [reselect](#) for a really great and concise implementation of composable selectors.

Baobab

[Baobab](#) is another popular library implementing immutable API for updating plain JavaScript objects. While you can use it with Redux, there is little benefit in using them together.

Most of the functionality Baobab provides is related to updating the data with cursors, but Redux enforces that the only way to update the data is to dispatch an **action**. Therefore they solve the same problem differently, and don't complement each other.

Unlike Immutable, Baobab doesn't yet implement any special efficient data structures under the hood, so you don't really win anything from using it together with Redux. It's easier to just use plain objects in this case.

RxJS

[RxJS](#) is a superb way to manage the complexity of asynchronous apps. In fact [there is an effort to create a library that models human-computer interaction as interdependent observables](#).

Does it make sense to use Redux together with RxJS? Sure! They work great together. For example, it is easy to expose a Redux **store** as an observable:

```
function toObservable(store) {
  return {
    subscribe({ next }) {
      const unsubscribe = store.subscribe(() => next(store.getState()))
      next(store.getState())
      return { unsubscribe }
    }
  }
}
```

Similarly, you can compose different asynchronous streams to turn them into actions before feeding them to `store.dispatch()`.

The question is: do you really need Redux if you already use Rx? Maybe not. It's not hard to [re-implement Redux in Rx](#). Some say it's a two-liner using Rx `.scan()` method. It may very well be!

If you're in doubt, check out the Redux source code (there isn't much going on there), as well as its ecosystem (for example, [the developer tools](#)). If you don't care too much about it and want to go with the reactive data flow all the way, you might want to explore something like [Cycle](#) instead, or even combine it with Redux. Let us know how it goes!

Learning Resources

The Redux docs are intended to teach the basic concepts of Redux, as well as explain key concepts for use in real-world applications. However, the docs can't cover everything. Happily, there are many other great resources available for learning Redux. We encourage you to check them out. Many of them cover topics that are beyond the scope of the docs, or describe the same topics in other ways that may work better for your learning style.

This page includes our recommendations for some of the best external resources available to learn Redux. For an additional extensive list of tutorials, articles, and other resources on React, Redux, Javascript, and related topics, see the [React/Redux Links list](#).

Basic Introductions

Tutorials that teach the basic concepts of Redux and how to use it

- **Getting Started with Redux - Video Series**

<https://egghead.io/series/getting-started-with-redux>

https://github.com/tayiorbeii/egghead.io_redux_course_notes

Dan Abramov, the creator of Redux, demonstrates various concepts in 30 short (2-5 minute) videos. The linked Github repo contains notes and transcriptions of the videos.

- **Building React Applications with Idiomatic Redux - Video Series**

<https://egghead.io/series/building-react-applications-with-idiomatic-redux>

https://github.com/tayiorbeii/egghead.io_idiomatic_redux_course_notes

Dan Abramov's second video tutorial series, continuing directly after the first. Includes lessons on [store](#) initial [state](#), using Redux with React Router, using "selector" functions, normalizing [state](#), use of Redux [middleware](#), async [action](#) creators, and more. The linked Github repo contains notes and transcriptions of the videos.

- **Live React: Hot Reloading and Time Travel**

<http://youtube.com/watch?v=xsSnOQynTHs>

Dan Abramov's original conference talk that introduced Redux. See how constraints enforced by Redux make hot reloading with time travel easy

- **A Cartoon Guide to Redux**

<https://code-cartoons.com/a-cartoon-intro-to-redux-3afb775501a6>

A high-level description of Redux, with friendly cartoons to help illustrate the ideas.

- **Leveling Up with React: Redux**

<https://css-tricks.com/learning-react-redux/>

A very well-written introduction to Redux and its related concepts, with some nifty cartoon-ish diagrams.

- **An Introduction to Redux**

<https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/>

An overview and intro to the basic concepts of Redux. Looks at the benefits of using Redux, how it differs from MVC or Flux, and its relation to functional programming.

- **Redux Tutorial**

<https://www.pshrmn.com/tutorials/react/redux/>

A short, clear tutorial that introduces basic Redux terms, shows how to split [reducer](#) functions, and describes

the Redux [store](#) API.

- **Redux: From Twitter Hype to Production**

<http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production#/>

An extremely well-produced slideshow that visually steps through core Redux concepts, usage with React, project organization, and side effects with thunks and sagas. Has some absolutely *fantastic* animated diagrams demonstrating how data flows through a React+Redux architecture.

- **DevGuides: Introduction to Redux**

<http://devguides.io/redux/>

A tutorial that covers several aspects of Redux, including actions, reducers, usage with React, and [middleware](#).

Using Redux With React

Explanations of the React-Redux bindings and the `connect` function

- **Why Redux is Useful in React Apps**

<https://www.fullstackreact.com/articles/redux-with-mark-erikson/>

An explanation of some of the benefits of using Redux with React, including sharing data between components and hot module reloading.

- **What Does Redux Do? (and when should you use it?)**

<https://daveceddia.com/what-does-redux-do/>

An excellent summary of how Redux helps solve data flow problems in a React app.

- **How Redux Works: A Counter-Example**

<https://daveceddia.com/how-does-redux-work/>

A great follow-up to the previous article. It explains how to use Redux and React-Redux, by first showing a React component that stores a value in its internal [state](#), and then refactoring it to use Redux instead. Along the way, the article explains important Redux terms and concepts, and how they all fit together to make the Redux data flow work properly.

- **Redux and React: An Introduction**

<http://jakesidsmith.com/blog/post/2017-11-18-redux-and-react-an-introduction/>

A great introduction to Redux's core concepts, with explanations of how to use the React-Redux package to use Redux with React.

Project-Based Tutorials

Tutorials that teach Redux concepts by building projects, including larger "real-world"-type applications

- **Practical Redux**

<http://blog.isquaredsoftware.com/2016/10/practical-redux-part-0-introduction/>

<http://blog.isquaredsoftware.com/series/practical-redux/>

An ongoing series of posts intended to demonstrate a number of specific Redux techniques by building a sample application, based on the MekHQ application for managing Battletech campaigns. Written by Redux co-maintainer Mark Erikson. Covers topics like managing relational data, connecting multiple components and lists, complex [reducer](#) logic for features, handling forms, showing modal dialogs, and much more.

- **Building a Simple CRUD App with React + Redux**

<http://www.thegreatcodeadventure.com/building-a-simple-crud-app-with-react-redux-part-1/>

A nifty 8-part series that demonstrates building a CRUD app, including routing, AJAX calls, and the various CRUD aspects. Very well written, with some useful diagrams as well.

- **The Soundcloud Client in React + Redux**

<http://www.robinwieruch.de/the-soundcloud-client-in-react-redux/>

A detailed walkthrough demonstrating project setup, routing, authentication, fetching of remote data, and wrapping of a stateful library.

- **Full-Stack Redux Tutorial**

<http://teropa.info/blog/2015/09/10/full-stack-redux-tutorial.html>

A full-blown, in-depth tutorial that builds up a complete client-server application.

- **Getting Started with React, Redux and Immutable: a Test-Driven Tutorial**

<http://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-1/>

<http://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-2/>

Another solid, in-depth tutorial, similar to the "Full-Stack" tutorial. Builds a client-only TodoMVC app, and demonstrates a good project setup (including a Mocha+JSDOM-based testing configuration). Well-written, covers many concepts, and very easy to follow.

- **Redux Hero: An Intro to Redux and Reselect**

<https://decembersoft.com/posts/redux-hero-part-1-a-hero-is-born-a-fun-introduction-to-redux-js/>

An introduction to Redux and related libraries through building a small RPG-style game

Redux Implementation

Explanations of how Redux works internally, by writing miniature reimplementations

- **Build Yourself a Redux**

<https://zapier.com/engineering/how-to-build-redux/>

An excellent in-depth "build a mini-Redux" article, which covers not only Redux's core, but also `connect` and `middleware` as well.

- **Connect.js explained**

<https://gist.github.com/gaearon/1d19088790e70ac32ea636c025ba424e>

A very simplified version of React Redux's `connect()` function that illustrates the basic implementation

- **Let's Write Redux!**

<http://www.jamasoftware.com/blog/lets-write-redux/>

Walks through writing a miniature version of Redux step-by-step, to help explain the concepts and implementation.

Reducers

Articles discussing ways to write `reducer` functions

- **Taking Advantage of `combineReducers`**
<http://randycoulman.com/blog/2016/11/22/taking-advantage-of-combinereducers/>
 Examples of using `combineReducers` multiple times to produce a `state` tree, and some thoughts on tradeoffs in various approaches to `reducer` logic.
- **The Power of Higher-Order Reducers**
<http://slides.com/omnidan/hor#/>
 A slideshow from the author of `redux-undo` and other libraries, explaining the concept of higher-order reducers and how they can be used
- **Reducer composition with Higher Order Reducers**
https://medium.com/@mange_vibration/reducer-composition-with-higher-order-reducers-35c3977ed08f
 Some great examples of writing small functions that can be composed together to perform larger specific `reducer` tasks, such as providing initial `state`, filtering, updating specific keys, and more.
- **Higher Order Reducers - It just sounds scary**
<https://medium.com/@danielkagan/high-order-reducers-it-just-sounds-scary-2b9e5dbfc705>
 Explains how reducers can be composed like Lego bricks to create reusable and testable `reducer` logic.

Selectors

Explanations of how and why to use selector functions to read values from `state`

- **Idiomatic Redux: Using Reselect Selectors for Encapsulation and Performance**
<https://blog.isquaredsoftware.com/2017/12/idiomatic-redux-using-reselect-selectors/>
 A complete guide to why you should use selector functions with Redux, how to use the Reselect library to write optimized selectors, and advanced tips for improving performance.
- **ReactCasts #8: Selectors in Redux**
<https://www.youtube.com/watch?v=frT3to2ACCw>
 A great overview of why and how to use selector functions to retrieve data from the `store`, and derive additional data from `store` values
- **Optimizing React Redux Application Development with Reselect**
<https://codebrahma.com/reselect-tutorial-optimizing-react-redux-application-development-with-reselect/>
 A good tutorial on Reselect. Covers the concept of "selector functions", how to use Reselect's API, and how to use memoized selectors to improve performance.
- **Usage of Reselect in a React-Redux Application**
<https://dashbouquet.com/blog/frontend-development/usage-of-reselect-in-a-react-redux-application>
 Discusses the importance of memoized selectors for performance, and good practices for using Reselect.
- **React, Reselect, and Redux**
<https://medium.com/@parkerdan/react-reselect-and-redux-b34017f8194c>
 An explanation of how Reselect's memoized selector functions are useful in Redux apps, and how to create unique selector instances for each component instance.

Normalization

How to structure the Redux `store` like a database for best performance

- **Querying a Redux Store**

<https://medium.com/@adamrackis/querying-a-redux-store-37db8c7f3b0f>

A look at best practices for organizing and storing data in Redux, including normalizing data and use of selector functions.

- **Normalizing Redux Stores for Maximum Code Reuse**

<https://medium.com/@adamrackis/normalizing-redux-stores-for-maximum-code-reuse-ae6e3844ae95>

Thoughts on how normalized Redux stores enable some useful data handling approaches, with examples of using selector functions to denormalize hierarchical data.

- **Advanced Redux Entity Normalization**

<https://medium.com/@dcousineau/advanced-redux-entity-normalization-f5f1fe2aefc5>

Describes a "keyWindow" concept for tracking subsets of entities in `state`, similar to an SQL "view". A useful extension to the idea of normalized data.

Middleware

Explanations and examples of how `middleware` work and how to write them

- **Exploring Redux Middlewares**

<http://blog.krawaller.se/posts/exploring-redux-middleware/>

Understanding middlewares through a series of small experiments

- **Redux Middleware Tutorial**

<http://www.pshrmn.com/tutorials/react/redux-middleware/>

An overview of what `middleware` is, how `applyMiddleware` works, and how to write `middleware`.

- **ReactCasts #6: Redux Middleware**

<https://www.youtube.com/watch?v=T-qtHI1qHlg>

A screencast that describes how `middleware` fit into Redux, their uses, and how to implement a custom `middleware`

- **A Beginner's Guide to Redux Middleware**

<https://www.codementor.io/reactjs/tutorial/beginner-s-guide-to-redux-middleware>

A useful explanation of `middleware` use cases, with numerous examples

- **Functional Composition in Javascript**

<https://joeortopassi.com/articles/functional-composition-in-javascript/>

Breaking down how the `compose` function works

Side Effects - Basics

Introductions to handling async behavior in Redux

- **Stack Overflow: Dispatching Redux Actions with a Timeout**

<http://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#35415559>

Dan Abramov explains the basics of managing async behavior in Redux, walking through a progressive series of approaches (inline async calls, async `action` creators, thunk `middleware`).

- **Stack Overflow: Why do we need [middleware](http://stackoverflow.com/questions/34570758/why-do-we-need-middleware-for-async-flow-in-redux/34599594#34599594) for async flow in Redux?**
<http://stackoverflow.com/questions/34570758/why-do-we-need-middleware-for-async-flow-in-redux/34599594#34599594>
 Dan Abramov gives reasons for using thunks and async [middleware](#), and some useful patterns for using thunks.
- **What the heck is a "thunk"?**
<https://daveceddia.com/what-is-a-thunk/>
 A quick explanation for what the word "thunk" means in general, and for Redux specifically.
- **Thunks in Redux: The Basics**
<https://medium.com/fullstack-academy/thunks-in-redux-the-basics-85e538a3fe60>
 A detailed look at what thunks are, what they solve, and how to use them.

Side Effects - Advanced

Advanced tools and techniques for managing async behavior

- **What is the right way to do asynchronous operations in Redux?**
<https://decembersoft.com/posts/what-is-the-right-way-to-do-asynchronous-operations-in-redux/>
 An excellent look at the most popular libraries for Redux side effects, with comparisons of how each one works.
- **Redux 4 Ways**
<https://medium.com/react-native-training/redux-4-ways-95a130da0cdc>
 Side-by-side comparisons of implementing some basic data fetching using thunks, sagas, observables, and a promise [middleware](#)
- **Idiomatic Redux: Thoughts on Thunks, Sagas, Abstractions, and Reusability**
<http://blog.isquaredsoftware.com/2017/01/idiomatic-redux-thoughts-on-thunks-sagas-abstraction-and-reusability/>
 A response to several "thunks are bad" concerns, arguing that thunks (and sagas) are still a valid approach for managing complex sync logic and async side effects.
- **Javascript Power Tools: Redux-Saga**
<http://formidable.com/blog/2017/javascript-power-tools-redux-saga/>
<http://formidable.com/blog/2017/composition-patterns-in-redux-saga/>
<http://formidable.com/blog/2017/real-world-redux-saga-patterns/>
 A fantastic series that teaches the concepts, implementation, and benefits behind Redux-Saga, including how ES6 generators are used to control function flow, how sagas can be composed together to accomplish concurrency, and practical use cases for sagas.
- **Exploring Redux Sagas**
<https://medium.com/onfido-tech/exploring-redux-sagas-cc1fca2015ee>
 An excellent article that explores how to use sagas to provide a glue layer to implement decoupled business logic in a Redux application.
- **Taming Redux with Sagas**
<https://objectpartners.com/2017/11/20/taming-redux-with-sagas/>
 A good overview of Redux-Saga, including info on generator functions, use cases for sagas, using sagas to deal with promises, and testing sagas.

- **Reactive Redux [State](#) with RxJS**

<https://ivanjov.com/reactive-redux-state-with-rxjs/>

Describes the concept of "Reactive Programming" and the RxJS library, and shows how to use redux-observable to fetch data, along with examples of testing.

- **Using redux-observable to handle asynchronous logic in Redux**

<https://medium.com/dailyjs/using-redux-observable-to-handle-asynchronous-logic-in-redux-d49194742522>

An extended post that compares a thunk-based implementation of handling a line-drawing example vs an observable-based implementation.

Thinking in Redux

Deeper looks at how Redux is meant to be used, and why it works the way it does

- **You Might Not Need Redux**

https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

Dan Abramov discusses the tradeoffs involved in using Redux.

- **Idiomatic Redux: The Tao of Redux, Part 1 - Implementation and Intent**

<http://blog.isquaredsoftware.com/2017/05/idiomatic-redux-tao-of-redux-part-1/>

A deep dive into how Redux actually works, the constraints it asks you to follow, and the intent behind its design and usage.

- **Idiomatic Redux: The Tao of Redux, Part 2 - Practice and Philosophy**

<http://blog.isquaredsoftware.com/2017/05/idiomatic-redux-tao-of-redux-part-2/>

A follow-up look at why common Redux usage patterns exist, other ways that Redux can be used, and thoughts on the pros and cons of those different patterns and approaches.

- **What's So Great About Redux?**

<https://medium.freecodecamp.org/whats-so-great-about-redux-ac16f1cc0f8b>

<https://storify.com/acemarke/redux-pros-cons-and-limitations>

<https://twitter.com/modernserf/status/886426115874717697>

Deep and fascinating analysis of how Redux compares to OOP and message-passing, how typical Redux usage can devolve towards Java-like "setter" functions with more boilerplate, and something of a plea for a higher-level "blessed" abstraction on top of Redux to make it easier to work with and learn for newbies. Very worth reading. The author originally wrote a tweetstorm, which is captured in the Storify link, and wrote the blog post to expand on those thoughts. Finally, he followed up with a few more thoughts on abstract vs concrete examples in another shorter tweet thread.

Redux Architecture

Patterns and practices for structuring larger Redux applications

- **Avoiding Accidental Complexity When Structuring Your App [State](#)**

<https://hackernoon.com/avoiding-accidental-complexity-when-structuring-your-app-state-6e6d22ad5e2a>

An excellent set of guidelines for organizing your Redux [store](#) structure.

- **Redux Step by Step: A Simple and Robust Workflow for Real Life Apps**

<https://hackernoon.com/redux-step-by-step-a-simple-and-robust-workflow-for-real-life-apps-1fdf7df46092>

A follow-up to the "Accidental Complexity" article, discussing principle

- **Things I Wish I Knew About Redux**

<https://medium.com/horrible-hacks/things-i-wish-i-knew-about-redux-9924abf2f9e0>

https://www.reddit.com/r/javascript/comments/4taau2/things_i_wish_i_knew_about_redux/

A number of excellent tips and lessons learned after building an app with Redux. Includes info on connecting components, selecting data, and app/project structure. Additional discussion on Reddit.

- **React+Redux: Tips and Best Practices for Clean, Reliable, & Maintainable Code**

<https://speakerdeck.com/goopscoop/react-plus-redux-tips-and-best-practices-for-clean-reliable-and-scalable-code>

An excellent slideshow with a wide variety of tips and suggestions, including keeping [action](#) creators simple and data manipulation in reducers, abstracting away API calls, avoiding spreading props, and more.

- **Redux for [state](#) management in large web apps**

<https://www.mapbox.com/blog/redux-for-state-management-in-large-web-apps/>

Excellent discussion and examples of idiomatic Redux architecture, and how Mapbox applies those approaches to their Mapbox Studio application.

Apps and Examples

- **React-Redux RealWorld Example: TodoMVC for the Real World**

<https://github.com/GoThinkster/redux-review>

An example full-stack "real world" application built with Redux. Demos a Medium-like social blogging site that includes JWT authentication, CRUD, favoriting articles, following users, routing, and more. The RealWorld project also includes many other implementations of the front and back ends of the site, specifically intended to show how different server and client implementations of the same project and API spec compare with each other.

- **Project Mini-Mek**

<https://github.com/markarikson/project-minimek>

A sample app to demonstrate various useful Redux techniques, accompanying the "Practical Redux" blog series at <http://blog.isquaredsoftware.com/series/practical-redux>

- **react-redux-yelp-clone**

<https://github.com/mohamed-ismat/react-redux-yelp-clone>

An adaptation of the "Yelp Clone" app by FullStackReact. It extends the original by using Redux and Redux Saga instead of local [state](#), as well as React Router v4, styled-components, and other modern standards. Based on the React-Boilerplate starter kit.

- **WordPress-Calypso**

<https://github.com/Automattic/wp-calypso>

The new JavaScript- and API-powered WordPress.com

- **Sound-Redux**

<https://github.com/andrewngu/sound-redux>

A Soundcloud client built with React / Redux

- **Webamp**

<https://webamp.org>

<https://github.com/captbaritone/webamp>

An in-browser recreation of Winamp2, built with React and Redux. Actually plays MP3s, and lets you load in local MP3 files.

- **Tello**
<https://github.com/joshwcomeau/Tello>
A simple and delightful way to track and manage TV shows
- **io-808**
<https://github.com/vincentriemer/io-808>
An attempt at a fully recreated web-based TR-808 drum machine

Redux Docs Translations

- [中文文档](#) — Chinese
- [繁體中文文件](#) — Traditional Chinese
- [Redux in Russian](#) — Russian
- [Redux en Español](#) - Spanish

Books

- **Redux in Action**
<https://www.manning.com/books/redux-in-action>
A comprehensive book that covers many key aspects of using Redux, including the basics of reducers and actions and use with React, complex middlewares and side effects, application structure, performance, testing, and much more. Does a great job of explaining the pros, cons, and tradeoffs of many approaches to using Redux. Personally recommended by Redux co-maintainer Mark Erikson.
- **The Complete Redux Book**
<https://leanpub.com/redux-book>
How do I manage a large [state](#) in production? Why do I need [store](#) enhancers? What is the best way to handle form validations? Get the answers to all these questions and many more using simple terms and sample code. Learn everything you need to use Redux to build complex and production-ready web applications. (Note: now permanently free!)
- **Taming the [State](#) in React**
<https://www.robinwieruch.de/learn-react-redux-mobx-state-management/>
If you have learned React with the previous book of the author called The Road to learn React, Taming the [State](#) in React will be the perfect blend to learn about basic and advanced [state](#) management in React. You will start out with learning only Redux without React. Afterward, the book shows you how to connect Redux to your React application. The advanced chapters will teach you about normalization, naming, selectors and asynchronous actions. In the end, you will set up and build a real world application with React and Redux.

Courses

- **Modern React with Redux, by Stephen Grider (paid)**
<https://www.udemy.com/react-redux/>
Master the fundamentals of React and Redux with this tutorial as you develop apps with React Router, Webpack, and ES6. This course will get you up and running quickly, and teach you the core knowledge you need to deeply understand and build React components and structure applications with Redux.

- **Redux, by Tyler McGinnis (paid)**

<https://tylermcginnis.com/courses/redux/>

When learning Redux, you need to learn it in the context of an app big enough to see the benefits. That's why this course is huge. A better name might be *"Real World Redux"*. If you're sick of "todo list" Redux tutorials, you've come to the right place. In this course we'll talk all about what makes Redux special for managing [state](#) in your application. We'll build an actual "real world" application so you can see how Redux handles edge cases like optimistic updates and error handling. We'll also cover many other technologies that work well with Redux, Firebase, and CSS Modules.

- **Learn Redux, by Wes Bos (free)**

<https://learnredux.com/>

A video course that walks through building 'Reduxstagram' — a simple photo app that will simplify the core ideas behind Redux, React Router and React.js

More Resources

- [React-Redux Links](#) is a curated list of high-quality articles, tutorials, and related content for React, Redux, ES6, and more.
- [Redux Ecosystem Links](#) is a categorized collection of Redux-related libraries, addons, and utilities.
- [Awesome Redux](#) is an extensive list of Redux-related repositories.
- [DEV Community](#) is a place to share Redux projects, articles and tutorials as well as start discussions and ask for feedback on Redux-related topics. Developers of all skill-levels are welcome to take part.

Ecosystem

Redux is a tiny library, but its contracts and APIs are carefully chosen to spawn an ecosystem of tools and extensions, and the community has created a wide variety of helpful addons, libraries, and tools. You don't need to use any of these addons to use Redux, but they can help make it easier to implement features and solve problems in your application.

For an extensive catalog of libraries, addons, and tools related to Redux, check out the [Redux Ecosystem Links](#) list. Also, the [React/Redux Links](#) list contains tutorials and other useful resources for anyone learning React or Redux.

This page lists some of the Redux-related addons that the Redux maintainers have vetted personally, or that have shown widespread adoption in the community. Don't let this discourage you from trying the rest of them! The ecosystem is growing too fast, and we have a limited time to look at everything. Consider these the “staff picks”, and don't hesitate to submit a PR if you've built something wonderful with Redux.

Table of Contents

- [Library Integration and Bindings](#)
- [Reducers](#)
 - [Reducer Combination](#)
 - [Reducer Composition](#)
 - [Higher-Order Reducers](#)
- [Actions](#)
- [Utilities](#)
- [Store](#)
 - [Change Subscriptions](#)
 - [Batching](#)
 - [Persistence](#)
- [Immutable Data](#)
 - [Data Structures](#)
 - [Immutable Update Utilities](#)
 - [Immutable/Redux Interop](#)
- [Side Effects](#)
 - [Widely Used](#)
 - [Promises](#)
- [Middleware](#)
 - [Networks and Sockets](#)
 - [Async Behavior](#)
 - [Analytics](#)
- [Entities and Collections](#)
- [Component State and Encapsulation](#)
- [Dev Tools](#)
 - [Debuggers and Viewers](#)
 - [DevTools Monitors](#)
 - [Logging](#)
 - [Mutation Detection](#)

- [Testing](#)
- [Routing](#)
- [Forms](#)
- [Higher-Level Abstractions](#)
- [Community Conventions](#)

Library Integration and Bindings

[reduxjs/react-redux](#)

The official React bindings for Redux, maintained by the Redux team

[angular-redux/ng-redux](#)

Angular 1 bindings for Redux

[angular-redux/store](#)

Angular 2+ bindings for Redux

[ember-redux/ember-redux](#)

Ember bindings for Redux

[glimmer-redux/glimmer-redux](#)

Redux bindings for Ember's Glimmer component engine

[tur-nr/polymer-redux](#)

Redux bindings for Polymer

[lastmjs/redux-store-element](#) Redux bindings for custom elements

Reducers

Reducer Combination

[ryo33/combineSectionReducers](#)

An expanded version of `combineReducers`, which allows passing `state` as a third argument to all slice reducers.

[KodersLab/topologically-combine-reducers](#)

A `combineReducers` variation that allows defining cross-slice dependencies for ordering and data passing

```
var masterReducer = topologicallyCombineReducers(
  { auth, users, todos },
  // define the dependency tree
  { auth: ['users'], todos: ['auth'] }
)
```

Reducer Composition

[acdlite/reduce-reducers](#)

Provides sequential composition of reducers at the same level

```
const combinedReducer = combineReducers({ users, posts, comments })
const rootReducer = reduceReducers(combinedReducer, otherTopLevelFeatureReducer)
```

[mhelmer/redux-xforms](#)

A collection of composable [reducer](#) transformers

```
const createByFilter = (predicate, mapActionToKey) =>
  compose(
    withInitialState({}), // inject initial state as {}
    withFilter(predicate), // let through if action has filterName
    updateSlice(mapActionToKey), // update a single key in the state
    isolateSlice(mapActionToKey) // run the reducer on a single state slice
  )
```

[adrienjt/redux-data-structures](#)

[Reducer](#) factory functions for common data structures: counters, maps, lists (queues, stacks), sets

```
const myCounter = counter({
  incrementActionTypes: ['INCREMENT'],
  decrementActionTypes: ['DECREMENT']
})
```

Higher-Order Reducers

[omnidan/redux-undo](#)

Effortless undo/redo and [action](#) history for your reducers

[omnidan/redux-ignore](#)

Ignore redux actions by array or filter function

[omnidan/redux-recycle](#)

Reset the redux [state](#) on certain actions

[ForbesLindesay/redux-optimist](#)

A [reducer](#) enhancer to enable type-agnostic optimistic updates

Actions

[reduxactions/redux-actions](#)

Flux Standard [Action](#) utilities for Redux

```
const increment = createAction('INCREMENT')
const reducer = handleActions({ [increment]: (state, action) => state + 1 }, 0)
const store = createStore(reducer)
store.dispatch(increment())
```

[BerkeleyTrue/redux-create-types](#)

Creates standard and async [action](#) types based on namespaces

```
export const types = createTypes(
  ['openModal', createAsyncTypes('fetch')],
  'app'
)
// { openModal : "app.openModal", fetch : { start : "app.fetch.start", complete: 'app.fetch.complete' } }
```

[maxhallinan/kreighter](#)

Generates [action](#) creators based on types and expected fields

```
const formatTitle = (id, title) => ({
  id,
  title: toTitleCase(title)
})
const updateBazTitle = fromType('UPDATE_BAZ_TITLE', formatTitle)
updateBazTitle(1, 'foo bar baz')
// -> { type: 'UPDATE_BAZ_TITLE', id: 1, title: 'Foo Bar Baz', }
```

Utilities

[reduxjs/reselect](#)

Creates composable memoized selector functions for efficiently deriving data from the [store state](#)

```
const taxSelector = createSelector(
  [subtotalSelector, taxPercentSelector],
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
)
```

[paularmstrong/normalizr](#)

Normalizes nested JSON according to a schema

```
const user = new schema.Entity('users')
const comment = new schema.Entity('comments', { commenter: user })
const article = new schema.Entity('articles', {
  author: user,
  comments: [comment]
})
const normalizedData = normalize(originalData, article)
```

[planttheidea/selectorator](#)

Abstractions over Reselect for common selector use cases

```
const getBarBaz = createSelector(
  ['foo.bar', 'baz'],
  (bar, baz) => `${bar} ${baz}`
)
getBarBaz({ foo: { bar: 'a' }, baz: 'b' }) // "a b"
```

Store

Change Subscriptions

[jprichardson/redux-watch](#)

Watch for [state](#) changes based on key paths or selectors

```
let w = watch(() => mySelector(store.getState()))
store.subscribe(
  w((newVal, oldVal) => {
    console.log(newVal, oldVal)
  })
)
```

```
)
```

[ashaffer/redux-subscribe](#)

Centralized subscriptions to [state](#) changes based on paths

```
store.dispatch( subscribe("users.byId.abcd", "subscription1", () => {} ) );
```

Batching

[tappleby/redux-batched-subscribe](#)

[Store](#) enhancer that can debounce subscription notifications

```
const debounceNotify = _.debounce(notify => notify())
const store = createStore(
  reducer,
  initialState,
  batchedSubscribe(debounceNotify)
)
```

[manaflair/redux-batch](#)

[Store](#) enhancer that allows dispatching arrays of actions

```
const store = createStore(reducer, reduxBatch)
store.dispatch([ { type: 'INCREMENT' }, { type: 'INCREMENT' } ])
```

[laysent/redux-batch-actions-enhancer](#)

[Store](#) enhancer that accepts batched actions

```
const store = createStore(reducer, initialState, batch().enhancer)
store.dispatch(createAction({ type: 'INCREMENT' }, { type: 'INCREMENT' } ))
```

[tshelburne/redux-batched-actions](#)

Higher-order [reducer](#) that handles batched actions

```
const store = createStore(enableBatching(reducer), initialState)
store.dispatch(batchActions([ { type: 'INCREMENT' }, { type: 'INCREMENT' } ]))
```

Persistence

[rt2zz/redux-persist](#)

Persist and rehydrate a Redux [store](#), with many extensible options

```
const store = createStore(reducer, autoRehydrate())
persistStore(store)
```

[react-stack/redux-storage](#)

Persistence layer for Redux with flexible backends

```
const reducer = storage.reducer(combineReducers(reducers))
const engine = createEngineLocalStorage('my-save-key')
const storageMiddleware = storage.createMiddleware(engine)
```

```
const store = createStore(reducer, applyMiddleware(storageMiddleware))
```

[redux-offline/redux-offline](#)

Persistent [store](#) for Offline-First apps, with support for optimistic UIs

```
const store = createStore(reducer, offline(offlineConfig))
store.dispatch({
  type: 'FOLLOW_USER_REQUEST',
  meta: { offline: { effect: {}, commit: {}, rollback: {} } }
})
```

Immutable Data

Data Structures

[facebook/immutable-js](#)

Immutable persistent data collections for Javascript

```
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 50)
map1.get('b') // 2
map2.get('b') // 50
```

[rtfeldman/seamless-immutable](#)

Frozen immutable arrays/objects, backwards-compatible with JS

```
const array = Immutable(['totally', 'immutable', { a: 42 }])
array[0] = 'edited' // does nothing
```

[planttheidea/crio](#)

Immutable JS objects with a natural API

```
const foo = crio(['foo'])
const fooBar = foo.push('bar') // new array: ['foo', 'bar']
```

[aearly/icepick](#)

Utilities for treating frozen JS objects as persistent immutable collections.

```
const newObj = icepick.assocIn({ c: { d: 'bar' } }, ['c', 'd'], 'baz')
const obj3 = icepicke.merge(obj1, obj2)
```

Immutable Update Utilities

[mweststrate/immer](#)

Immutable updates with normal mutative code, using Proxies

```
const nextState = produce(baseState, draftState => {
  draftState.push({ todo: 'Tweet about it' })
  draftState[1].done = true
})
```

[kolodny/immutability-helper](#)

A drop-in replacement for react-addons-update

```
const newData = update(myData, {
  x: { y: { z: { $set: 7 } } },
  a: { b: { $push: [9] } }
})
```

[mariocasciaro/object-path-immutable](#)

Simpler alternative to immutability-helpers and Immutable.js

```
const newObj = immutable(obj)
  .set('a.b', 'f')
  .del(['a', 'c', 0])
  .value()
```

[debitoor/dot-prop-immutable](#)

Immutable version of the dot-prop lib, with some extensions

```
const newState = dotProp.set(state, `todos.${index}.complete`, true)
const endOfArray = dotProp.get(obj, 'foo.$end')
```

Immutable/Redux Interop

[gajus/redux-immutable](#)

combineReducers equivalent that works with Immutable.js Maps

```
const initialState = Immutable.Map()
const rootReducer = combineReducers({})
const store = createStore(rootReducer, initialState)
```

[eadmundo/redux-seamless-immutable](#)

combineReducers equivalent that works with seamless-immutable values

```
import { combineReducers } from 'redux-seamless-immutable';
const rootReducer = combineReducers({ userReducer, posts
```

Side Effects

Widely Used

[gaeareon/redux-thunk](#)

Dispatch functions, which are called and given `dispatch` and `getState` as parameters. This acts as a loophole for AJAX calls and other async behavior.

Best for: getting started, simple async and complex synchronous logic.

```
function fetchData(someValue) {
  return (dispatch, getState) => {
    dispatch({type: "REQUEST_STARTED"});

    myAjaxLib.post("/someEndpoint", {data: someValue})
```

```

        .then(response => dispatch({type : "REQUEST_SUCCEEDED", payload : response}))
        .catch(error => dispatch({type : "REQUEST_FAILED", error : error}));
    };
}

function addTodosIfAllowed(todoText) {
    return (dispatch, getState) => {
        const state = getState();

        if(state.todos.length < MAX_TODOs) {
            dispatch({type : "ADD_TODO", text : todoText});
        }
    }
}

```

redux-saga/redux-saga

Handle async logic using synchronous-looking generator functions. Sagas return descriptions of effects, which are executed by the saga [middleware](#), and act like "background threads" for JS applications.

Best for: complex async logic, decoupled workflows

```

function* fetchData(action) {
    const { someValue } = action
    try {
        const response = yield call(myAjaxLib.post, '/someEndpoint', {
            data: someValue
        })
        yield put({ type: 'REQUEST_SUCCEEDED', payload: response })
    } catch (error) {
        yield put({ type: 'REQUEST_FAILED', error: error })
    }
}

function* addTodosIfAllowed(action) {
    const { todoText } = action
    const todos = yield select(state => state.todos)

    if (todos.length < MAX_TODOs) {
        yield put({ type: 'ADD_TODO', text: todoText })
    }
}

```

redux-observable/redux-observable

Handle async logic using RxJS observable chains called "epics". Compose and cancel async actions to create side effects and more.

Best for: complex async logic, decoupled workflows

```

const loginRequestEpic = action$ =>
    action$
        .ofType(LOGIN_REQUEST)
        .mergeMap(({ payload: { username, password } }) =>
            Observable.from(postLogin(username, password))
                .map(loginSuccess)
                .catch(loginFailure)
        )

const loginSuccessfulEpic = action$ =>
    action$
        .ofType(LOGIN_SUCCESS)
        .delay(2000)

```



```
.mergeMap(({ payload: { msg } }) => showMessage(msg))

const rootEpic = combineEpics(loginRequestEpic, loginSuccessfulEpic)
```

redux-loop/redux-loop

A port of the Elm Architecture to Redux that allows you to sequence your effects naturally and purely by returning them from your reducers. Reducers now return both a [state](#) value and a side effect description.

Best for: trying to be as much like Elm as possible in Redux+JS

```
export const reducer = (state = {}, action) => {
  switch (action.type) {
    case ActionType.LOGIN_REQUEST:
      const { username, password } = action.payload
      return loop(
        { pending: true },
        Effect.promise(loginPromise, username, password)
      )
    case ActionType.LOGIN_SUCCESS:
      const { user, msg } = action.payload
      return loop(
        { pending: false, user },
        Effect.promise(delayMessagePromise, msg, 2000)
      )
    case ActionType.LOGIN_FAILURE:
      return { pending: false, err: action.payload }
    default:
      return state
  }
}
```

jeffbski/redux-logic

Side effects lib built with observables, but allows use of callbacks, promises, async/await, or observables. Provides declarative processing of actions.

Best for: very decoupled async logic

```
const loginLogic = createLogic({
  type: Actions.LOGIN_REQUEST,

  process({ getState, action }, dispatch, done) {
    const { username, password } = action.payload

    postLogin(username, password)
      .then(
        ({ user, msg }) => {
          dispatch(loginSucceeded(user))

          setTimeout(() => dispatch(showMessage(msg)), 2000)
        },
        err => dispatch(loginFailure(err))
      )
      .then(done)
  }
})
```

Promises

[acdlite/redux-promise](#)

Dispatch promises as [action](#) payloads, and have FSA-compliant actions dispatched as the promise resolves or rejects.

```
dispatch({ type: 'FETCH_DATA', payload: myAjaxLib.get('/data') })  
// will dispatch either {type: "FETCH_DATA", payload: response} if resolved,  
// or dispatch {type: "FETCH_DATA", payload: error, error: true} if rejected
```

[lelandrichardson/redux-pack](#)

Sensible, declarative, convention-based promise handling that guides users in a good direction without exposing the full power of dispatch.

```
dispatch({type: "FETCH_DATA", payload: myAjaxLib.get("/data") });  
  
// in a reducer:  
case "FETCH_DATA": =  
  return handle(state, action, {  
    start: prevState => ({  
      ...prevState,  
      isLoading: true,  
      fooError: null  
    }),  
    finish: prevState => ({ ...prevState, isLoading: false }),  
    failure: prevState => ({ ...prevState, fooError: payload }),  
    success: prevState => ({ ...prevState, foo: payload }),  
  });
```

Middleware

Networks and Sockets

[svrcekrmichal/redux-axios-middleware](#)

Fetches data with Axios and dispatches start/success/fail actions

```
export const loadCategories() => ({ type: 'LOAD', payload: { request: { url: '/categories' } } });
```

[agraboso/redux-api-middleware](#)

Reads API call actions, fetches, and dispatches FSAs

```
const fetchUsers = () => ({  
  [CALL_API]: {  
    endpoint: 'http://www.example.com/api/users',  
    method: 'GET',  
    types: ['REQUEST', 'SUCCESS', 'FAILURE']  
  }  
})
```

[itaylor/redux-socket.io](#)

An opinionated connector between socket.io and redux.

```
const store = createStore(reducer, applyMiddleware(socketIoMiddleware))  
store.dispatch({ type: 'server/hello', data: 'Hello!' })
```

[tiberiuc/redux-react-firebase](#)

Integration between Firebase, React, and Redux

Async Behavior

[rt2zz/redux-action-buffer](#)

Buffers all actions into a queue until a breaker condition is met, at which point the queue is released

[wyze/redux-debounce](#)

FSA-compliant [middleware](#) for Redux to debounce actions.

[mathieudutour/redux-queue-offline](#)

Queue actions when offline and dispatch them when getting back online.

Analytics

[rangle/redux-beacon](#)

Integrates with any analytics services, can track while offline, and decouples analytics logic from app logic

[hyperlab/redux-insights](#)

Analytics and tracking with an easy API for writing your own adapters

[markdalglish/redux-analytics](#)

Watches for Flux Standard Actions with meta analytics values and processes them

Entities and Collections

[tommikaikkonen/redux-orm](#)

A simple immutable ORM to manage relational data in your Redux [store](#).

[Versent/redux-crud](#)

Convention-based actions and reducers for CRUD logic

[kwelch/entities-reducer](#)

A higher-order [reducer](#) that handles data from Normalizr

[amplitude/redux-query](#)

Declare colocated data dependencies with your components, run queries when components mount, perform optimistic updates, and trigger server changes with Redux actions.

[cantierecreativo/redux-bees](#)

Declarative JSON-API interaction that normalizes data, with a React HOC that can run queries

[GetAmbassador/redux-clerk](#)

Async CRUD handling with normalization, optimistic updates, sync/async [action](#) creators, selectors, and an extendable [reducer](#).

[shoutem/redux-io](#)

JSON-API abstraction with async CRUD, normalization, optimistic updates, caching, data status, and error handling.

[jmeas/redux-resource](#)

A tiny but powerful system for managing 'resources': data that is persisted to remote servers.

Component State and Encapsulation

[tonyhb/redux-ui](#)

"Block-level scoping" for UI [state](#). Decorated components declare data fields, which become props and can be updated by nested children.

```
@ui({
  key: 'some-name',
  state: { uiVar1: '', uiVar2: (props, state) => state.someValue },
  reducer: (state, action) => {}
})
class YourComponent extends React.Component {}
```

[threepointone/redux-react-local](#)

Local component [state](#) in Redux, with handling for component actions

```
@local({
  ident: 'counter', initial: 0, reducer : (state, action) => action.me ? state + 1 : state }
})
class Counter extends React.Component {
```

[epeli/lean-redux](#)

Makes component [state](#) in Redux as easy as `setState`

```
const DynamicCounters = connectLean(
  scope: "dynamicCounters",
  getInitialState() => ({counterCount : 1}),
  addCounter, removeCounter
)(CounterList);
```

[ioof-holdings/redux-subspace](#)

Creates isolated "sub-stores" for decoupled micro front-ends, with integration for React, sagas, and observables

```
const reducer = combineReducers({
  subApp1: namespaced('subApp1')(counter),
  subApp2: namespaced('subApp2')(counter)
})

const subApp1Store = subspace(state => state.subApp1, 'subApp1')(store)
const subApp2Store = subspace(state => state.subApp2, 'subApp2')(store)

subApp1Store.dispatch({ type: 'INCREMENT' })
console.log('store state:', store.getState()) // { "subApp1": { value: 2 }, "subApp2": { value: 1 } }
```

[DataDog/redux-doghouse](#)

Aims to make reusable components easier to build with Redux by scoping actions and reducers to a particular instance of a component.

```
const scopeableActions = new ScopedActionFactory(actionCreators)
const actionCreatorsScopedToA = scopeableActions.scope('a')
actionCreatorsScopedToA.foo('bar') //{ type: SET_F00, value: 'bar', scopeID: 'a' }

const boundScopeableActions = bindScopedActionFactories(
  scopeableActions,
  store.dispatch
)
```

```
const scopedReducers = scopeReducers(reducers)
```

Dev Tools

Debuggers and Viewers

[reduxjs/redux-devtools](#)

Dan Abramov's original Redux DevTools implementation, built for in-app display of [state](#) and time-travel debugging

[zalmoxisus/redux-devtools-extension](#)

Mihail Diordiev's browser extension, which bundles multiple [state](#) monitor views and adds integration with the browser's own dev tools

[infinitered/reactotron](#)

A cross-platform Electron app for inspecting React and React Native apps, including app [state](#), API requests, perf, errors, sagas, and [action](#) dispatching.

DevTools Monitors

[Log Monitor](#)

The default monitor for Redux DevTools with a tree view

[Dock Monitor](#)

A resizable and movable dock for Redux DevTools monitors

[Slider Monitor](#)

A custom monitor for Redux DevTools to replay recorded Redux actions

[Inspector](#)

A custom monitor for Redux DevTools that lets you filter actions, inspect diffs, and pin deep paths in the [state](#) to observe their changes

[Diff Monitor](#)

A monitor for Redux DevTools that diffs the Redux [store](#) mutations between actions

[Filterable Log Monitor](#)

Filterable tree view monitor for Redux DevTools

[Chart Monitor](#)

A chart monitor for Redux DevTools

[Filter Actions](#)

Redux DevTools composable monitor with the ability to filter actions

Logging

[evgenyrodionov/redux-logger](#)

Logging [middleware](#) that shows actions, states, and diffs

[inakianduaga/redux-state-history](#)

Enhancer that provides time-travel and efficient [action](#) recording capabilities, including import/export of [action](#) logs and [action](#) playback.

[joshwcomeau/redux-vcr](#)

Record and replay user sessions in real-time

[socialtables/redux-unhandled-action](#)

Warns about actions that produced no [state](#) changes in development

Mutation Detection

[leoasis/redux-immutable-state-invariant](#)

[Middleware](#) that throws an error when you try to mutate your [state](#) either inside a dispatch or between dispatches.

[flexport/mutation-sentinel](#)

Helps you deeply detect mutations at runtime and enforce immutability in your codebase.

[mmahalwy/redux-pure-connect](#)

Check and log whether react-redux's connect method is passed `mapState` functions that create impure props.

Testing

[arnaudbenard/redux-mock-store](#)

A mock [store](#) that saves dispatched actions in an array for assertions

[Workable/redux-test-belt](#)

Extends the [store](#) API to make it easier assert, isolate, and manipulate the [store](#)

[conorhastings/redux-test-recorder](#)

[Middleware](#) to automatically generate reducers tests based on actions in the app

[wix/redux-testkit](#)

Complete and opinionated testkit for testing Redux projects (reducers, selectors, actions, thanks)

[jfairbank/redux-saga-test-plan](#)

Makes integration and unit testing of sagas a breeze

Routing

[supasate/connected-react-router](#) Synchronize React Router 4 [state](#) with your Redux [store](#).

[FormidableLabs/redux-little-router](#)

A tiny router for Redux applications that lets the URL do the talking

[faceyspacey/redux-first-router](#)

Seamless Redux-first routing. Think of your app in states, not routes, not components, while keeping the address bar in sync. Everything is [state](#). Connect your components and just dispatch flux standard actions.

Forms

[erikras/redux-form](#)

A full-featured library to enable a React HTML form to [store](#) its [state](#) in Redux.

[davidkpiano/react-redux-form](#)

React Redux Form is a collection of [reducer](#) creators and [action](#) creators that make implementing even the most complex and custom forms with React and Redux simple and performant.

Higher-Level Abstractions

[keajs/kea](#)

An abstraction over Redux, Redux-Saga and Reselect. Provides a framework for your app's actions, reducers, selectors and sagas. It empowers Redux, making it as simple to use as `setState`. It reduces boilerplate and redundancy, while retaining composability.

[jumpsuit/jumpstate](#)

A simplified layer over Redux. No [action](#) creators or explicit dispatching, with a built-in simple side effects system.

[TheComfyChair/redux-scc](#)

Takes a defined structure and uses 'behaviors' to create a set of actions, [reducer](#) responses and selectors.

[Bloomca/redux-tiles](#)

Provides minimal abstraction on top of Redux, to allow easy composability, easy async requests, and sane testability.

Community Conventions

[Flux Standard Action](#)

A human-friendly standard for Flux [action](#) objects

[Canonical Reducer Composition](#)

An opinionated standard for nested [reducer](#) composition

[Ducks: Redux Reducer Bundles](#)

A proposal for bundling reducers, [action](#) types and actions

Examples

Redux is distributed with a few examples in its [source code](#). Most of these examples are also on [CodeSandbox](#), an online editor that lets you play with the examples online.

Counter Vanilla

Run the [Counter Vanilla](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/counter-vanilla  
open index.html
```

Or check out the [sandbox](#):



It does not require a build system or a view framework and exists to show the raw Redux API used with ES5.

Counter

Run the [Counter](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/counter  
npm install  
npm start
```

Or check out the [sandbox](#):



This is the most basic example of using Redux together with React. For simplicity, it re-renders the React component manually when the [store](#) changes. In real projects, you will likely want to use the highly performant [React Redux](#) bindings instead.

This example includes tests.

Todos

Run the [Todos](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/todos  
npm install  
npm start
```

Or check out the [sandbox](#):



This is the best example to get a deeper understanding of how the [state](#) updates work together with components in Redux. It shows how reducers can delegate handling actions to other reducers, and how you can use [React Redux](#) to generate container components from your presentational components.

This example includes tests.

Todos with Undo

Run the [Todos with Undo](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/todos-with-undo  
npm install  
npm start
```

Or check out the [sandbox](#):



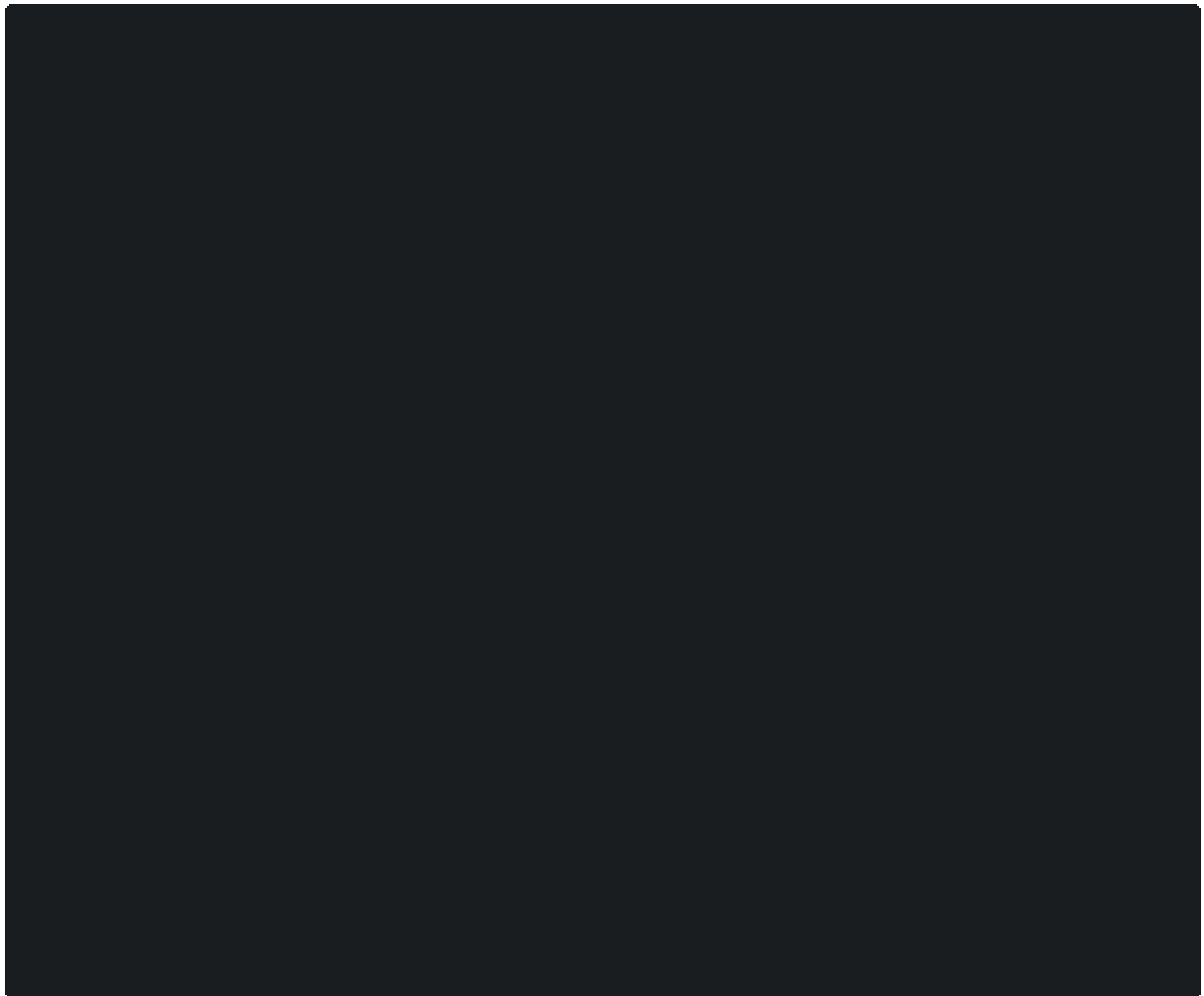
This is a variation on the previous example. It is almost identical, but additionally shows how wrapping your [reducer](#) with [Redux Undo](#) lets you add a Undo/Redo functionality to your app with a few lines of code.

Todos w/ Flow

Run the [Todos w/ Flow](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/todos-flow  
npm install  
npm start
```

Or check out the [sandbox](#):



This is like the previous Todos examples, but shows how to use Redux in conjunction with [Flow](#).

TodoMVC

Run the [TodoMVC](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/todomvc  
npm install  
npm start
```

Or check out the [sandbox](#):



This is the classical [TodoMVC](#) example. It's here for the sake of comparison, but it covers the same points as the Todos example.

This example includes tests.

Shopping Cart

Run the [Shopping Cart](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/shopping-cart  
npm install  
npm start
```

Or check out the [sandbox](#):



This example shows important idiomatic Redux patterns that become important as your app grows. In particular, it shows how to [store](#) entities in a normalized way by their IDs, how to compose reducers on several levels, and how to define selectors alongside the reducers so the knowledge about the [state](#) shape is encapsulated. It also demonstrates logging with [Redux Logger](#) and conditional dispatching of actions with [Redux Thunk middleware](#).

Tree View

Run the [Tree View](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/tree-view  
npm install  
npm start
```

Or check out the [sandbox](#):



This example demonstrates rendering a deeply nested tree view and representing its [state](#) in a normalized form so it is easy to update from reducers. Good rendering performance is achieved by the container components granularly subscribing only to the tree nodes that they render.

This example includes tests.

Async

Run the [Async](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/async  
npm install  
npm start
```

Or check out the [sandbox](#):



This example includes reading from an asynchronous API, fetching data in response to user input, showing loading indicators, caching the response, and invalidating the cache. It uses [Redux Thunk middleware](#) to encapsulate asynchronous side effects.

Universal

Run the [Universal](#) example:

```
git clone https://github.com/reduxjs/redux.git  
  
cd redux/examples/universal  
npm install  
npm start
```

This is a basic demonstration of [server rendering](#) with Redux and React. It shows how to prepare the initial [store state](#) on the server, and pass it down to the client so the client [store](#) can boot up from an existing [state](#).

Real World

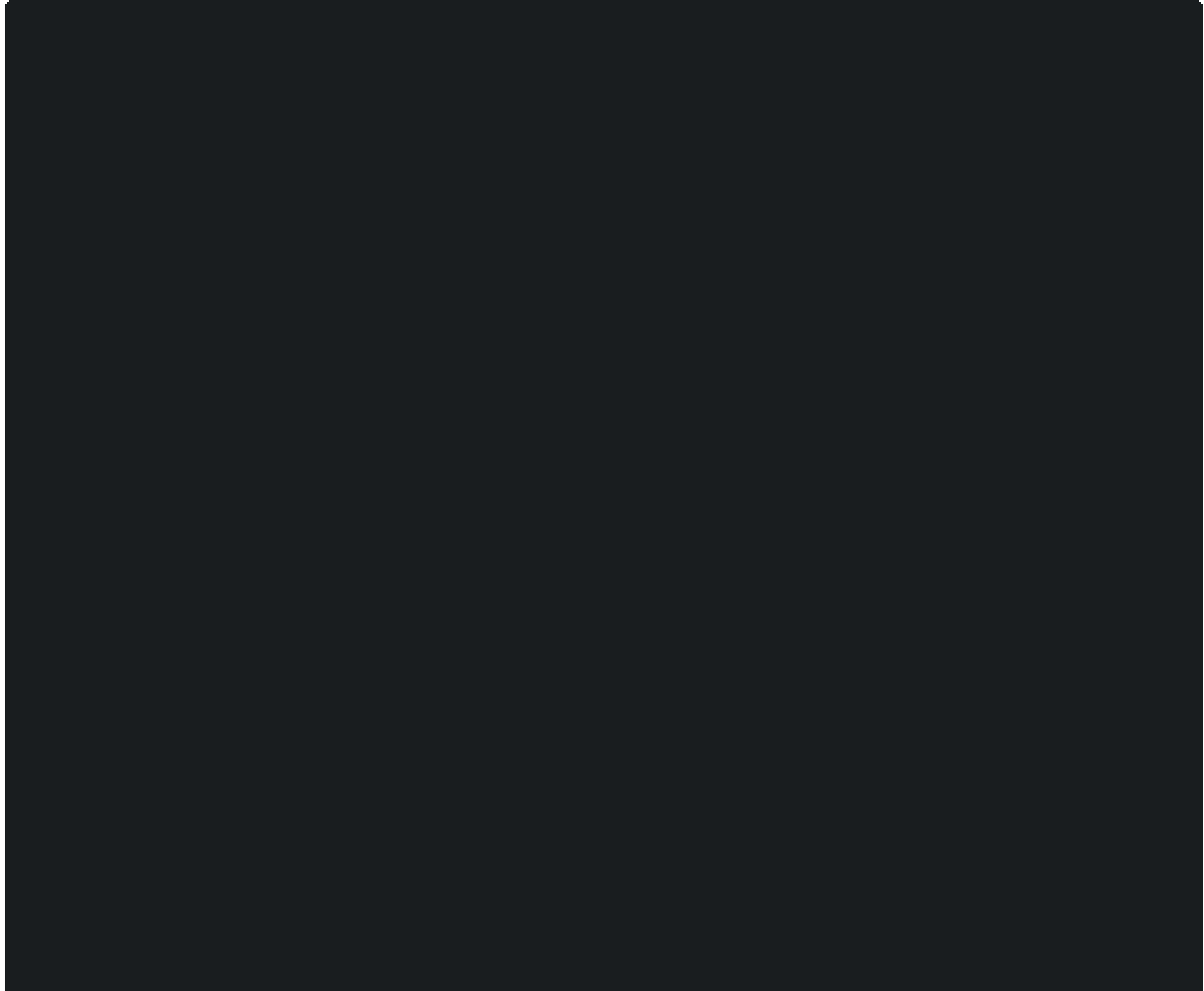
Run the [Real World](#) example:

```
git clone https://github.com/reduxjs/redux.git
```



```
cd redux/examples/real-world
npm install
npm start
```

Or check out the [sandbox](#):



This is the most advanced example. It is dense by design. It covers keeping fetched entities in a normalized cache, implementing a custom [middleware](#) for API calls, rendering partially loaded data, pagination, caching responses, displaying error messages, and routing. Additionally, it includes Redux DevTools.

More Examples

You can find more examples in the [Redux Apps and Examples](#) page of the [Redux Addons Catalog](#).

Basics

Don't be fooled by all the fancy talk about reducers, [middleware](#), [store](#) enhancers—Redux is incredibly simple. If you've ever built a Flux application, you will feel right at home. If you're new to Flux, it's easy too!

In this guide, we'll walk through the process of creating a simple Todo app.

- [Actions](#)
- [Reducers](#)
- [Store](#)
- [Data Flow](#)
- [Usage with React](#)
- [Example: Todo List](#)

Actions

First, let's define some actions.

Actions are payloads of information that send data from your application to your [store](#). They are the *only* source of information for the [store](#). You send them to the [store](#) using `store.dispatch()`.

Here's an example [action](#) which represents adding a new todo item:

```
const ADD_TODO = 'ADD_TODO'
```

```
{
  type: ADD_TODO,
  text: 'Build my first Redux app'
}
```

Actions are plain JavaScript objects. Actions must have a `type` property that indicates the type of [action](#) being performed. Types should typically be defined as string constants. Once your app is large enough, you may want to move them into a separate module.

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes'
```

Note on Boilerplate

You don't have to define [action](#) type constants in a separate file, or even to define them at all. For a small project, it might be easier to just use string literals for [action](#) types. However, there are some benefits to explicitly declaring constants in larger codebases. Read [Reducing Boilerplate](#) for more practical tips on keeping your codebase clean.

Other than `type`, the structure of an [action](#) object is really up to you. If you're interested, check out [Flux Standard Action](#) for recommendations on how actions could be constructed.

We'll add one more [action](#) type to describe a user ticking off a todo as completed. We refer to a particular todo by `index` because we [store](#) them in an array. In a real app, it is wiser to generate a unique ID every time something new is created.

```
{
  type: TOGGLE_TODO,
  index: 5
}
```

It's a good idea to pass as little data in each [action](#) as possible. For example, it's better to pass `index` than the whole todo object.

Finally, we'll add one more [action](#) type for changing the currently visible todos.

```
{
  type: SET_VISIBILITY_FILTER,
  filter: SHOW_COMPLETED
}
```

Action Creators

Action creators are exactly that—functions that create actions. It's easy to conflate the terms “[action](#)” and “[action creator](#)”, so do your best to use the proper term.

In Redux, [action](#) creators simply return an [action](#):

```
function addTodo(text) {  
  return {  
    type: ADD_TODO,  
    text  
  }  
}
```

This makes them portable and easy to test.

In [traditional Flux](#), [action](#) creators often trigger a dispatch when invoked, like so:

```
function addTodoWithDispatch(text) {  
  const action = {  
    type: ADD_TODO,  
    text  
  }  
  dispatch(action)  
}
```

In Redux this is *not* the case.

Instead, to actually initiate a dispatch, pass the result to the `dispatch()` function:

```
dispatch(addTodo(text))  
dispatch(completeTodo(index))
```

Alternatively, you can create a **bound [action creator](#)** that automatically dispatches:

```
const boundAddTodo = text => dispatch(addTodo(text))  
const boundCompleteTodo = index => dispatch(completeTodo(index))
```

Now you'll be able to call them directly:

```
boundAddTodo(text)  
boundCompleteTodo(index)
```

The `dispatch()` function can be accessed directly from the [store](#) as `store.dispatch()`, but more likely you'll access it using a helper like [react-redux](#)'s `connect()`. You can use `bindActionCreators()` to automatically bind many [action](#) creators to a `dispatch()` function.

[Action](#) creators can also be asynchronous and have side-effects. You can read about [async actions](#) in the [advanced tutorial](#) to learn how to handle AJAX responses and compose [action](#) creators into async control flow. Don't skip ahead to async actions until you've completed the basics tutorial, as it covers other important concepts that are prerequisite for the advanced tutorial and async actions.

Source Code

actions.js

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * other constants
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * action creators
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function toggleTodo(index) {
  return { type: TOGGLE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

Next Steps

Now let's [define some reducers](#) to specify how the [state](#) updates when you dispatch these actions!

Reducers

Reducers specify how the application's [state](#) changes in response to [actions](#) sent to the [store](#). Remember that actions only describe *what happened*, but don't describe how the application's [state](#) changes.

Designing the State Shape

In Redux, all the application [state](#) is stored as a single object. It's a good idea to think of its shape before writing any code. What's the minimal representation of your app's [state](#) as an object?

For our todo app, we want to [store](#) two different things:

- The currently selected visibility filter.
- The actual list of todos.

You'll often find that you need to [store](#) some data, as well as some UI [state](#), in the [state](#) tree. This is fine, but try to keep the data separate from the UI [state](#).

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

Note on Relationships

In a more complex app, you're going to want different entities to reference each other. We suggest that you keep your [state](#) as normalized as possible, without any nesting. Keep every entity in an object stored with an ID as a key, and use IDs to reference it from other entities, or lists. Think of the app's [state](#) as a database. This approach is described in [normalizr's](#) documentation in detail. For example, keeping `todosById: { id -> todo }` and `todos: array<id>` inside the [state](#) would be a better idea in a real app, but we're keeping the example simple.

Handling Actions

Now that we've decided what our [state](#) object looks like, we're ready to write a [reducer](#) for it. The [reducer](#) is a pure function that takes the previous [state](#) and an [action](#), and returns the next [state](#).

```
(previousState, action) => newState
```

It's called a [reducer](#) because it's the type of function you would pass to `Array.prototype.reduce(reducer, ? initialValue)`. It's very important that the [reducer](#) stays pure. Things you should **never** do inside a [reducer](#):

- Mutate its arguments;
- Perform side effects like API calls and routing transitions;
- Call non-pure functions, e.g. `Date.now()` Or `Math.random()` .

We'll explore how to perform side effects in the [advanced walkthrough](#). For now, just remember that the **reducer** must be pure. **Given the same arguments, it should calculate the next `state` and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.**

With this out of the way, let's start writing our **reducer** by gradually teaching it to understand the **actions** we defined earlier.

We'll start by specifying the initial **state**. Redux will call our **reducer** with an `undefined` **state** for the first time. This is our chance to return the initial **state** of our app:

```
import { VisibilityFilters } from './actions'

const initialState = {
  visibilityFilter: VisibilityFilters.SHOW_ALL,
  todos: []
}

function todoApp(state, action) {
  if (typeof state === 'undefined') {
    return initialState
  }

  // For now, don't handle any actions
  // and just return the state given to us.
  return state
}
```

One neat trick is to use the [ES6 default arguments syntax](#) to write this in a more compact way:

```
function todoApp(state = initialState, action) {
  // For now, don't handle any actions
  // and just return the state given to us.
  return state
}
```

Now let's handle `SET_VISIBILITY_FILTER` . All it needs to do is to change `visibilityFilter` on the **state**. Easy:

```
import {
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'

...

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Note that:

1. **We don't mutate the `state`** . We create a copy with `Object.assign()` . `Object.assign(state, { visibilityFilter: action.filter })` is also wrong: it will mutate the first argument. You **must** supply an empty object as the first parameter. You can also enable the [object spread operator proposal](#) to write `{ ...state, ...newState }` instead.
2. **We return the previous `state` in the `default` case**. It's important to return the previous `state` for any unknown [action](#).

Note on `Object.assign`

`Object.assign()` is a part of ES6, and is not supported by older browsers. To support them, you will need to either use a polyfill, a [Babel plugin](#), or a helper from another library like `_assign()` .

Note on `switch` and Boilerplate

The `switch` statement is *not* the real boilerplate. The real boilerplate of Flux is conceptual: the need to emit an update, the need to register the [Store](#) with a Dispatcher, the need for the [Store](#) to be an object (and the complications that arise when you want a universal app). Redux solves these problems by using pure reducers instead of event emitters.

It's unfortunate that many still choose a framework based on whether it uses `switch` statements in the documentation. If you don't like `switch` , you can use a custom `createReducer` function that accepts a handler map, as shown in ["reducing boilerplate"](#) .

Handling More Actions

We have two more actions to handle! Just like we did with `SET_VISIBILITY_FILTER` , we'll import the `ADD_TODO` and `TOGGLE_TODO` actions and then extend our [reducer](#) to handle `ADD_TODO` .

```
import {
  ADD_TODO,
  TOGGLE_TODO,
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'

...

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    default:
      return state
  }
}
```



```

    }
  }
}

```

Just like before, we never write directly to `state` or its fields, and instead we return new objects. The new `todos` is equal to the old `todos` concatenated with a single new item at the end. The fresh todo was constructed using the data from the [action](#).

Finally, the implementation of the `TOGGLE_TODO` handler shouldn't come as a complete surprise:

```

case TOGGLE_TODO:
  return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
      if (index === action.index) {
        return Object.assign({}, todo, {
          completed: !todo.completed
        })
      }
      return todo
    })
  })
}

```

Because we want to update a specific item in the array without resorting to mutations, we have to create a new array with the same items except the item at the index. If you find yourself often writing such operations, it's a good idea to use a helper like [immutability-helper](#), [updeep](#), or even a library like [Immutable](#) that has native support for deep updates. Just remember to never assign to anything inside the `state` unless you clone it first.

Splitting Reducers

Here is our code so far. It is rather verbose:

```

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: [
          ...state.todos,
          {
            text: action.text,
            completed: false
          }
        ]
      })
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: state.todos.map((todo, index) => {
          if (index === action.index) {
            return Object.assign({}, todo, {
              completed: !todo.completed
            })
          }
          return todo
        })
      })
    default:
      return state
  }
}

```

```

}
}

```

Is there a way to make it easier to comprehend? It seems like `todos` and `visibilityFilter` are updated completely independently. Sometimes `state` fields depend on one another and more consideration is required, but in our case we can easily split updating `todos` into a separate function:

```

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    case ADD_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}

```

Note that `todos` also accepts `state` —but `state` is an array! Now `todoApp` gives `todos` just a slice of the `state` to manage, and `todos` knows how to update just that slice. **This is called *reducer composition*, and it's the fundamental pattern of building Redux apps.**

Let's explore `reducer` composition more. Can we also extract a `reducer` managing just `visibilityFilter`? We can.

Below our imports, let's use [ES6 Object Destructuring](#) to declare `SHOW_ALL`:

```

const { SHOW_ALL } = VisibilityFilters

```

Then:

```
function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}
```

Now we can rewrite the main **reducer** as a function that calls the reducers managing parts of the **state**, and combines them into a single object. It also doesn't need to know the complete initial **state** any more. It's enough that the child reducers return their initial **state** when given `undefined` at first.

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Note that each of these reducers is managing its own part of the global **state. The `state` parameter is different for every **reducer**, and corresponds to the part of the **state** it manages.**

This is already looking good! When the app is larger, we can split the reducers into separate files and keep them completely independent and managing different data domains.

Finally, Redux provides a utility called `combineReducers()` that does the same boilerplate logic that the `todoApp` above currently does. With its help, we can rewrite `todoApp` like this:

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Note that this is equivalent to:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

You could also give them different keys, or call functions differently. These two ways to write a combined `reducer` are equivalent:

```
const reducer = combineReducers({
  a: doSomethingWithA,
  b: processB,
  c: c
})
```

```
function reducer(state = {}, action) {
  return {
    a: doSomethingWithA(state.a, action),
    b: processB(state.b, action),
    c: c(state.c, action)
  }
}
```

All `combineReducers()` does is generate a function that calls your reducers **with the slices of `state` selected according to their keys**, and combines their results into a single object again. *It's not magic*. And like other reducers, `combineReducers()` does not create a new object if all of the reducers provided to it do not change `state`.

Note for ES6 Savvy Users

Because `combineReducers` expects an object, we can put all top-level reducers into a separate file, `export` each `reducer` function, and use `import * as reducers` to get them as an object with their names as the keys:

```
import { combineReducers } from 'redux'
import * as reducers from './reducers'

const todoApp = combineReducers(reducers)
```

Because `import *` is still new syntax, we no longer use it in the documentation to avoid `confusion`, but you may encounter it in some community examples.

Source Code

reducers.js

```
import { combineReducers } from 'redux'
import {
  ADD_TODO,
  TOGGLE_TODO,
  SET_VISIBILITY_FILTER,
  VisibilityFilters
} from './actions'
const { SHOW_ALL } = VisibilityFilters

function visibilityFilter(state = SHOW_ALL, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case TOGGLE_TODO:
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        }
        return todo
      })
    default:
      return state
  }
}

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Next Steps

Next, we'll explore how to [create a Redux store](#) that holds the [state](#) and takes care of calling your [reducer](#) when you dispatch an [action](#).

Store

In the previous sections, we defined the [actions](#) that represent the facts about “what happened” and the [reducers](#) that update the [state](#) according to those actions.

The [Store](#) is the object that brings them together. The [store](#) has the following responsibilities:

- Holds application [state](#);
- Allows access to [state](#) via `getState()` ;
- Allows [state](#) to be updated via `dispatch(action)` ;
- Registers listeners via `subscribe(listener)` ;
- Handles unregistering of listeners via the function returned by `subscribe(listener)` .

It's important to note that you'll only have a single [store](#) in a Redux application. When you want to split your data handling logic, you'll use [reducer composition](#) instead of many stores.

It's easy to create a [store](#) if you have a [reducer](#). In the [previous section](#), we used `combineReducers()` to combine several reducers into one. We will now import it, and pass it to `createStore()` .

```
import { createStore } from 'redux'
import todoApp from './reducers'
const store = createStore(todoApp)
```

You may optionally specify the initial [state](#) as the second argument to `createStore()` . This is useful for hydrating the [state](#) of the client to match the [state](#) of a Redux application running on the server.

```
const store = createStore(todoApp, window.STATE_FROM_SERVER)
```

Dispatching Actions

Now that we have created a [store](#), let's verify our program works! Even without any UI, we can already test the update logic.

```
import {
  addTodo,
  toggleTodo,
  setVisibilityFilter,
  VisibilityFilters
} from './actions'

// Log the initial state
console.log(store.getState())

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
const unsubscribe = store.subscribe(() => console.log(store.getState()))

// Dispatch some actions
store.dispatch(addTodo('Learn about actions'))
store.dispatch(addTodo('Learn about reducers'))
store.dispatch(addTodo('Learn about store'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
```

```
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))

// Stop listening to state updates
unsubscribe()
```

You can see how this causes the [state](#) held by the [store](#) to change:

```
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[0]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[1]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[2]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▶ Object {visibleTodoFilter: "SHOW_ALL", todos: Array[3]}
▼ Object {visibleTodoFilter: "SHOW_COMPLETED", todos: Array[3]} ⓘ
  ▼ todos: Array[3]
    ▼ 0: Object
      completed: true
      text: "Learn about actions"
      ▶ __proto__: Object
    ▼ 1: Object
      completed: true
      text: "Learn about reducers"
      ▶ __proto__: Object
    ▼ 2: Object
      completed: false
      text: "Learn about store"
      ▶ __proto__: Object
      length: 3
      ▶ __proto__: Array[0]
      visibleTodoFilter: "SHOW_COMPLETED"
      ▶ __proto__: Object
```

We specified the behavior of our app before we even started writing the UI. We won't do this in this tutorial, but at this point you can write tests for your reducers and [action](#) creators. You won't need to mock anything because they are just [pure](#) functions. Call them, and make assertions on what they return.

Source Code

index.js

```
import { createStore } from 'redux'
import todoApp from './reducers'

const store = createStore(todoApp)
```

Next Steps

Before creating a UI for our todo app, we will take a detour to see [how the data flows in a Redux application](#).

Data Flow

Redux architecture revolves around a **strict unidirectional data flow**.

This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand. It also encourages data normalization, so that you don't end up with multiple, independent copies of the same data that are unaware of one another.

If you're still not convinced, read [Motivation](#) and [The Case for Flux](#) for a compelling argument in favor of unidirectional data flow. Although [Redux is not exactly Flux](#), it shares the same key benefits.

The data lifecycle in any Redux app follows these 4 steps:

1. You call `store.dispatch(action)` .

An [action](#) is a plain object describing *what happened*. For example:

```
{ type: 'LIKE_ARTICLE', articleId: 42 }  
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }  
{ type: 'ADD_TODO', text: 'Read the Redux docs.' }
```

Think of an [action](#) as a very brief snippet of news. “Mary liked article 42.” or “‘Read the Redux docs.’ was added to the list of todos.”

You can call `store.dispatch(action)` from anywhere in your app, including components and XHR callbacks, or even at scheduled intervals.

1. The Redux [store](#) calls the [reducer](#) function you gave it.

The [store](#) will pass two arguments to the [reducer](#): the current [state](#) tree and the [action](#). For example, in the todo app, the root [reducer](#) might receive something like this:

```
// The current application state (list of todos and chosen filter)  
let previousState = {  
  visibleTodoFilter: 'SHOW_ALL',  
  todos: [  
    {  
      text: 'Read the docs.',  
      complete: false  
    }  
  ]  
}  
  
// The action being performed (adding a todo)  
let action = {  
  type: 'ADD_TODO',  
  text: 'Understand the flow.'  
}  
  
// Your reducer returns the next application state  
let nextState = todoApp(previousState, action)
```

Note that a [reducer](#) is a pure function. It only *computes* the next [state](#). It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an [action](#) is dispatched.

1. The root **reducer** may combine the output of multiple reducers into a single **state** tree.

How you structure the root **reducer** is completely up to you. Redux ships with a `combineReducers()` helper function, useful for “splitting” the root **reducer** into separate functions that each manage one branch of the **state** tree.

Here's how `combineReducers()` works. Let's say you have two reducers, one for a list of todos, and another for the currently selected filter setting:

```
function todos(state = [], action) {
  // Somehow calculate it...
  return nextState
}

function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // Somehow calculate it...
  return nextState
}

let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

When you emit an **action**, `todoApp` returned by `combineReducers` will call both reducers:

```
let nextTodos = todos(state.todos, action)
let nextVisibleTodoFilter = visibleTodoFilter(state.visibleTodoFilter, action)
```

It will then combine both sets of results into a single **state** tree:

```
return {
  todos: nextTodos,
  visibleTodoFilter: nextVisibleTodoFilter
}
```

While `combineReducers()` is a handy helper utility, you don't have to use it; feel free to write your own root **reducer**!

1. The Redux **store** saves the complete **state** tree returned by the root **reducer**.

This new tree is now the next **state** of your app! Every listener registered with `store.subscribe(listener)` will now be invoked; listeners may call `store.getState()` to get the current **state**.

Now, the UI can be updated to reflect the new **state**. If you use bindings like **React Redux**, this is the point at which `component.setState(newState)` is called.

Next Steps

Now that you know how Redux works, let's [connect it to a React app](#).

Note for Advanced Users

If you're already familiar with the basic concepts and have previously completed this tutorial, don't forget to check out [async flow](#) in the [advanced tutorial](#) to learn how [middleware](#) transforms [async actions](#) before they reach the **reducer**.

Usage with React

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux works especially well with libraries like [React](#) and [Deku](#) because they let you describe UI as a function of [state](#), and Redux emits [state](#) updates in response to actions.

We will use React to build our simple todo app, and cover the basics of how to use React with Redux.

Note: see the [official React-Redux docs at https://react-redux.js.org](https://react-redux.js.org) for a complete guide on how to use Redux and React together.

Installing React Redux

[React bindings](#) are not included in Redux by default. You need to install them explicitly:

```
npm install --save react-redux
```

If you don't use npm, you may grab the latest UMD build from unpkg (either a [development](#) or a [production](#) build). The UMD build exports a global called `window.ReactRedux` if you add it to your page via a `<script>` tag.

Presentational and Container Components

React bindings for Redux separate *presentational* components from *container* components. This approach can make your app easier to understand and allow you to more easily reuse components. Here's a summary of the differences between presentational and container components (but if you're unfamiliar, we recommend that you also read [Dan Abramov's original article describing the concept of presentational and container components](#)):

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, state updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch Redux actions
Are written	By hand	Usually generated by React Redux

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux [store](#). This and the design brief below do not imply container components must be near the top of the component tree. If a container component becomes too complex (i.e. it has heavily nested presentational components with countless callbacks being passed down), introduce another container within the component tree as noted in the [FAQ](#).

Technically you could write the container components by hand using `store.subscribe()`. We don't advise you to do this because React Redux makes many performance optimizations that are hard to do by hand. For this reason, rather than write container components, we will generate them using the `connect()` function provided by

React Redux, as you will see below.

Designing Component Hierarchy

Remember how we [designed the shape of the root state object](#)? It's time we design the UI hierarchy to match it. This is not a Redux-specific task. [Thinking in React](#) is a great tutorial that explains the process.

Our design brief is simple. We want to show a list of todo items. On click, a todo item is crossed out as completed. We want to show a field where the user may add a new todo. In the footer, we want to show a toggle to show all, only completed, or only active todos.

Designing Presentational Components

I see the following presentational components and their props emerge from this brief:

- `TodoList` is a list showing visible todos.
 - `todos: Array` is an array of todo items with `{ id, text, completed }` shape.
 - `onTodoClick(id: number)` is a callback to invoke when a todo is clicked.
- `Todo` is a single todo item.
 - `text: string` is the text to show.
 - `completed: boolean` is whether the todo should appear crossed out.
 - `onClick()` is a callback to invoke when the todo is clicked.
- `Link` is a link with a callback.
 - `onClick()` is a callback to invoke when the link is clicked.
- `Footer` is where we let the user change currently visible todos.
- `App` is the root component that renders everything else.

They describe the *look* but don't know *where* the data comes from, or *how* to change it. They only render what's given to them. If you migrate from Redux to something else, you'll be able to keep all these components exactly the same. They have no dependency on Redux.

Designing Container Components

We will also need some container components to connect the presentational components to Redux. For example, the presentational `TodoList` component needs a container like `VisibleTodoList` that subscribes to the Redux [store](#) and knows how to apply the current visibility filter. To change the visibility filter, we will provide a `FilterLink` container component that renders a `Link` that dispatches an appropriate [action](#) on click:

- `VisibleTodoList` filters the todos according to the current visibility filter and renders a `TodoList`.
- `FilterLink` gets the current visibility filter and renders a `Link`.
 - `filter: string` is the visibility filter it represents.

Designing Other Components

Sometimes it's hard to tell if some component should be a presentational component or a container. For example, sometimes form and function are really coupled together, such as in the case of this tiny component:

- `AddTodo` is an input field with an "Add" button

Technically we could split it into two components but it might be too early at this stage. It's fine to mix presentation and logic in a component that is very small. As it grows, it will be more obvious how to split it, so we'll leave it mixed.

Implementing Components

Let's write the components! We begin with the presentational components so we don't need to think about binding to Redux yet.

Implementing Presentational Components

These are all normal React components, so we won't examine them in detail. We write functional stateless components unless we need to use local [state](#) or the lifecycle methods. This doesn't mean that presentational components *have to* be functions—it's just easier to define them this way. If and when you need to add local [state](#), lifecycle methods, or performance optimizations, you can convert them to classes.

components/ToDo.js

```
import React from 'react'
import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

components/ToDoList.js

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './ToDo'

const ToDoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={index} {...todo} onClick={() => onTodoClick(index)} />
    ))}
  </ul>
)

ToDoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
```

```

    id: PropTypes.number.isRequired,
    completed: PropTypes.bool.isRequired,
    text: PropTypes.string.isRequired
  }).isRequired
).isRequired,
onTodoClick: PropTypes.func.isRequired
}

export default TodoList

```

components/Link.js

```

import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a
      href=""
      onClick={e => {
        e.preventDefault()
        onClick()
      }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link

```

components/Footer.js

```

import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {' ', ' '}
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {' ', ' '}
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)

export default Footer

```

Implementing Container Components

Now it's time to hook up those presentational components to Redux by creating some containers. Technically, a container component is just a React component that uses `store.subscribe()` to read a part of the Redux `state` tree and supply props to a presentational component it renders. You could write a container component by hand, but we suggest instead generating container components with the React Redux library's `connect()` function, which provides many useful optimizations to prevent unnecessary re-renders. (One result of this is that you shouldn't have to worry about the [React performance suggestion](#) of implementing `shouldComponentUpdate` yourself.)

To use `connect()`, you need to define a special function called `mapStateToProps` that describes how to transform the current Redux `store state` into the props you want to pass to a presentational component you are wrapping. For example, `VisibleTodoList` needs to calculate `todos` to pass to the `TodoList`, so we define a function that filters the `state.todos` according to the `state.visibilityFilter`, and use it in its `mapStateToProps`:

```
const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
    case 'SHOW_ALL':
    default:
      return todos
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

In addition to reading the `state`, container components can dispatch actions. In a similar fashion, you can define a function called `mapDispatchToProps()` that receives the `dispatch()` method and returns callback props that you want to inject into the presentational component. For example, we want the `VisibleTodoList` to inject a prop called `onTodoClick` into the `TodoList` component, and we want `onTodoClick` to dispatch a `TOGGLE_TODO` `action`:

```
const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}
```

Finally, we create the `VisibleTodoList` by calling `connect()` and passing these two functions:

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

These are the basics of the React Redux API, but there are a few shortcuts and power options so we encourage you to check out [its documentation](#) in detail. In case you are worried about `mapStateToProps` creating new objects too often, you might want to learn about [computing derived data](#) with [reselect](#).

Find the rest of the container components defined below:

containers/FilterLink.js

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => {
  return {
    active: ownProps.filter === state.visibilityFilter
  }
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onClick: () => {
      dispatch(setVisibilityFilter(ownProps.filter))
    }
  }
}

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}
```

```

}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList

```

Implementing Other Components

containers/AddTodo.js

Recall as [mentioned previously](#), both the presentation and logic for the `AddTodo` component are mixed into a single definition.

```

import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

let AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form
        onSubmit={e => {
          e.preventDefault()
          if (!input.value.trim()) {
            return
          }
          dispatch(addTodo(input.value))
          input.value = ''
        }}
      >
        <input
          ref={node => {
            input = node
          }}
        />
        <button type="submit">Add Todo</button>
      </form>
    </div>
  )
}
AddTodo = connect()(AddTodo)

export default AddTodo

```

If you are unfamiliar with the `ref` attribute, please read this [documentation](#) to familiarize yourself with the recommended use of this attribute.

Tying the containers together within a component

components/App.js

```

import React from 'react'
import Footer from '../Footer'
import AddTodo from '../containers/AddTodo'

```

```
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App
```

Passing the Store

All container components need access to the Redux [store](#) so they can subscribe to it. One option would be to pass it as a prop to every container component. However it gets tedious, as you have to wire `store` even through presentational components just because they happen to render a container deep in the component tree.

The option we recommend is to use a special React Redux component called `<Provider>` to [magically](#) make the [store](#) available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Next Steps

Read the [complete source code for this tutorial](#) to better internalize the knowledge you have gained. Then, head straight to the [advanced tutorial](#) to learn how to handle network requests and routing!

You should also take some time to [read through the React-Redux docs](#) to get a better understanding of how to use React and Redux together.

Example: Todo List

This is the complete source code of the tiny todo app we built during the [basics tutorial](#). This code is also in [our repository of examples](#) and can be [run in your browser via CodeSandbox](#).

Entry Point

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import rootReducer from './reducers'
import App from './components/App'

const store = createStore(rootReducer)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Action Creators

actions/index.js

```
let nextTodoId = 0
export const addTodo = text => ({
  type: 'ADD_TODO',
  id: nextTodoId++,
  text
})

export const setVisibilityFilter = filter => ({
  type: 'SET_VISIBILITY_FILTER',
  filter
})

export const toggleTodo = id => ({
  type: 'TOGGLE_TODO',
  id
})

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}
```

Reducers

reducers/todos.js

```
const todos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          id: action.id,
          text: action.text,
          completed: false
        }
      ]
    case 'TOGGLE_TODO':
      return state.map(todo =>
        todo.id === action.id ? { ...todo, completed: !todo.completed } : todo
      )
    default:
      return state
  }
}

export default todos
```

reducers/visibilityFilter.js

```
import { VisibilityFilters } from '../actions'

const visibilityFilter = (state = VisibilityFilters.SHOW_ALL, action) => {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

export default visibilityFilter
```

reducers/index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

export default combineReducers({
  todos,
  visibilityFilter
})
```

Presentational Components

components/ToDo.js

```
import React from 'react'
import PropTypes from 'prop-types'
```

```

const Todo = ({ onClick, completed, text }) => (
  <li
    onClick={onClick}
    style={{
      textDecoration: completed ? 'line-through' : 'none'
    }}
  >
    {text}
  </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo

```

components/TodoList.js

```

import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, toggleTodo }) => (
  <ul>
    {todos.map(todo => (
      <Todo key={todo.id} {...todo} onClick={() => toggleTodo(todo.id)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
      text: PropTypes.string.isRequired
    }).isRequired
  ).isRequired,
  toggleTodo: PropTypes.func.isRequired
}

export default TodoList

```

components/Link.js

```

import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => (
  <button
    onClick={onClick}
    disabled={active}
    style={{
      marginLeft: '4px'
    }}
  >
    {children}
  </button>
)

```

```

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link

```

components/Footer.js

```

import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <div>
    <span>Show: </span>
    <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </div>
)

export default Footer

```

components/App.js

```

import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)

export default App

```

Container Components

containers/VisibleTodoList.js

```

import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { VisibilityFilters } from '../actions'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(t => t.completed)
  }
}

```



```

    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(t => !t.completed)
    default:
      throw new Error('Unknown filter: ' + filter)
  }
}

const mapStateToProps = state => ({
  todos: getVisibleTodos(state.todos, state.visibilityFilter)
})

const mapDispatchToProps = dispatch => ({
  toggleTodo: id => dispatch(toggleTodo(id))
})

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

```

containers/FilterLink.js

```

import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({
  active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
})

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

```

Other Components

containers/AddTodo.js

```

import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

const AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form
        onSubmit={e => {
          e.preventDefault()
          if (!input.value.trim()) {
            return
          }
          dispatch(addTodo(input.value))
          input.value = ''
        }}
      >

```

```
    >
      <input ref={node => (input = node)} />
      <button type="submit">Add Todo</button>
    </form>
  </div>
)
}

export default connect()(AddTodo)
```

Advanced

In the [basics walkthrough](#), we explored how to structure a simple Redux application. In this walkthrough, we will explore how AJAX and routing fit into the picture.

- [Async Actions](#)
- [Async Flow](#)
- [Middleware](#)
- [Usage with React Router](#)
- [Example: Reddit API](#)
- [Next Steps](#)

Async Actions

In the [basics guide](#), we built a simple todo application. It was fully synchronous. Every time an [action](#) was dispatched, the [state](#) was updated immediately.

In this guide, we will build a different, asynchronous application. It will use the Reddit API to show the current headlines for a selected subreddit. How does asynchronicity fit into Redux flow?

Actions

When you call an asynchronous API, there are two crucial moments in time: the moment you start the call, and the moment when you receive an answer (or a timeout).

Each of these two moments usually require a change in the application [state](#); to do that, you need to dispatch normal actions that will be processed by reducers synchronously. Usually, for any API request you'll want to dispatch at least three different kinds of actions:

- An [action](#) informing the reducers that the request began.

The reducers may handle this [action](#) by toggling an `isFetching` flag in the [state](#). This way the UI knows it's time to show a spinner.

- An [action](#) informing the reducers that the request finished successfully.

The reducers may handle this [action](#) by merging the new data into the [state](#) they manage and resetting `isFetching`. The UI would hide the spinner, and display the fetched data.

- An [action](#) informing the reducers that the request failed.

The reducers may handle this [action](#) by resetting `isFetching`. Additionally, some reducers may want to [store](#) the error message so the UI can display it.

You may use a dedicated `status` field in your actions:

```
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' }
{ type: 'FETCH_POSTS', status: 'success', response: { ... } }
```

Or you can define separate types for them:

```
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', error: 'Oops' }
{ type: 'FETCH_POSTS_SUCCESS', response: { ... } }
```

Choosing whether to use a single [action](#) type with flags, or multiple [action](#) types, is up to you. It's a convention you need to decide with your team. Multiple types leave less room for a mistake, but this is not an issue if you generate [action](#) creators and reducers with a helper library like [redux-actions](#).

Whatever convention you choose, stick with it throughout the application.

We'll use separate types in this tutorial.

Synchronous Action Creators

Let's start by defining the several synchronous [action](#) types and [action](#) creators we need in our example app. Here, the user can select a subreddit to display:

actions.js (Synchronous)

```
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT'

export function selectSubreddit(subreddit) {
  return {
    type: SELECT_SUBREDDIT,
    subreddit
  }
}
```

They can also press a “refresh” button to update it:

```
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'

export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT,
    subreddit
  }
}
```

These were the actions governed by the user interaction. We will also have another kind of [action](#), governed by the network requests. We will see how to dispatch them later, but for now, we just want to define them.

When it's time to fetch the posts for some subreddit, we will dispatch a `REQUEST_POSTS` [action](#):

```
export const REQUEST_POSTS = 'REQUEST_POSTS'

function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}
```

It is important for it to be separate from `SELECT_SUBREDDIT` or `INVALIDATE_SUBREDDIT`. While they may occur one after another, as the app grows more complex, you might want to fetch some data independently of the user [action](#) (for example, to prefetch the most popular subreddits, or to refresh stale data once in a while). You may also want to fetch in response to a route change, so it's not wise to couple fetching to some particular UI event early on.

Finally, when the network request comes through, we will dispatch `RECEIVE_POSTS`:

```
export const RECEIVE_POSTS = 'RECEIVE_POSTS'

function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}
```

```
}  
}
```

This is all we need to know for now. The particular mechanism to dispatch these actions together with network requests will be discussed later.

Note on Error Handling

In a real app, you'd also want to dispatch an [action](#) on request failure. We won't implement error handling in this tutorial, but the [real world example](#) shows one of the possible approaches.

Designing the State Shape

Just like in the basic tutorial, you'll need to [design the shape of your application's state](#) before rushing into the implementation. With asynchronous code, there is more [state](#) to take care of, so we need to think it through.

This part is often confusing to beginners, because it is not immediately clear what information describes the [state](#) of an asynchronous application, and how to organize it in a single tree.

We'll start with the most common use case: lists. Web applications often show lists of things. For example, a list of posts, or a list of friends. You'll need to figure out what sorts of lists your app can show. You want to [store](#) them separately in the [state](#), because this way you can cache them and only fetch again if necessary.

Here's what the [state](#) shape for our “Reddit headlines” app might look like:

```
{  
  selectedSubreddit: 'frontend',  
  postsBySubreddit: {  
    frontend: {  
      isFetching: true,  
      didInvalidate: false,  
      items: []  
    },  
    reactjs: {  
      isFetching: false,  
      didInvalidate: false,  
      lastUpdated: 1439478405547,  
      items: [  
        {  
          id: 42,  
          title: 'Confusion about Flux and Relay'  
        },  
        {  
          id: 500,  
          title: 'Creating a Simple Application Using React JS and Flux Architecture'  
        }  
      ]  
    }  
  }  
}
```

There are a few important bits here:

- We [store](#) each subreddit's information separately so we can cache every subreddit. When the user switches between them the second time, the update will be instant, and we won't need to refetch unless we want to. Don't worry about all these items being in memory: unless you're dealing with tens of thousands of items, and your user rarely closes the tab, you won't need any sort of cleanup.

- For every list of items, you'll want to `store isFetching` to show a spinner, `didInvalidate` so you can later toggle it when the data is stale, `lastUpdated` so you know when it was fetched the last time, and the `items` themselves. In a real app, you'll also want to `store pagination state` like `fetchPageCount` and `nextPageUrl` .

Note on Nested Entities

In this example, we `store` the received items together with the pagination information. However, this approach won't work well if you have nested entities referencing each other, or if you let the user edit items. Imagine the user wants to edit a fetched post, but this post is duplicated in several places in the `state` tree. This would be really painful to implement.

If you have nested entities, or if you let users edit received entities, you should keep them separately in the `state` as if it was a database. In pagination information, you would only refer to them by their IDs. This lets you always keep them up to date. The [real world example](#) shows this approach, together with `normalizr` to normalize the nested API responses. With this approach, your `state` might look like this:

```
{
  selectedSubreddit: 'frontend',
  entities: {
    users: {
      2: {
        id: 2,
        name: 'Andrew'
      }
    },
    posts: {
      42: {
        id: 42,
        title: 'Confusion about Flux and Relay',
        author: 2
      },
      100: {
        id: 100,
        title: 'Creating a Simple Application Using React JS and Flux Architecture',
        author: 2
      }
    }
  },
  postsBySubreddit: {
    frontend: {
      isFetching: true,
      didInvalidate: false,
      items: []
    },
    reactjs: {
      isFetching: false,
      didInvalidate: false,
      lastUpdated: 1439478405547,
      items: [ 42, 100 ]
    }
  }
}
```

In this guide, we won't normalize entities, but it's something you should consider for a more dynamic application.

Handling Actions

Before going into the details of dispatching actions together with network requests, we will write the reducers for the actions we defined above.

Note on Reducer Composition

Here, we assume that you understand [reducer](#) composition with `combineReducers()`, as described in the [Splitting Reducers](#) section on the [basics guide](#). If you don't, please [read it first](#).

reducers.js

```
import { combineReducers } from 'redux'
import {
  SELECT_SUBREDDIT,
  INVALIDATE_SUBREDDIT,
  REQUEST_POSTS,
  RECEIVE_POSTS
} from '../actions'

function selectedSubreddit(state = 'reactjs', action) {
  switch (action.type) {
    case SELECT_SUBREDDIT:
      return action.subreddit
    default:
      return state
  }
}

function posts(
  state = {
    isFetching: false,
    didInvalidate: false,
    items: []
  },
  action
) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
      return Object.assign({}, state, {
        didInvalidate: true
      })
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        isFetching: true,
        didInvalidate: false
      })
    case RECEIVE_POSTS:
      return Object.assign({}, state, {
        isFetching: false,
        didInvalidate: false,
        items: action.posts,
        lastUpdated: action.receivedAt
      })
    default:
      return state
  }
}

function postsBySubreddit(state = {}, action) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
    case RECEIVE_POSTS:
    case REQUEST_POSTS:
      return Object.assign({}, state, {
```



```

    [action.subreddit]: posts(state[action.subreddit], action)
  })
  default:
    return state
}
}

const rootReducer = combineReducers({
  postsBySubreddit,
  selectedSubreddit
})

export default rootReducer

```

In this code, there are two interesting parts:

- We use ES6 computed property syntax so we can update `state[action.subreddit]` with `Object.assign()` in a concise way. This:

```

return Object.assign({}, state, {
  [action.subreddit]: posts(state[action.subreddit], action)
})

```

is equivalent to this:

```

let nextState = {}
nextState[action.subreddit] = posts(state[action.subreddit], action)
return Object.assign({}, state, nextState)

```

- We extracted `posts(state, action)` that manages the [state](#) of a specific post list. This is just [reducer composition](#)! It is our choice how to split the [reducer](#) into smaller reducers, and in this case, we're delegating updating items inside an object to a `posts` [reducer](#). The [real world example](#) goes even further, showing how to create a [reducer](#) factory for parameterized pagination reducers.

Remember that reducers are just functions, so you can use functional composition and higher-order functions as much as you feel comfortable.

Async Action Creators

Finally, how do we use the synchronous [action](#) creators we [defined earlier](#) together with network requests? The standard way to do it with Redux is to use the [Redux Thunk middleware](#). It comes in a separate package called `redux-thunk`. We'll explain how [middleware](#) works in general [later](#); for now, there is just one important thing you need to know: by using this specific [middleware](#), an [action](#) creator can return a function instead of an [action](#) object. This way, the [action](#) creator becomes a [thunk](#).

When an [action](#) creator returns a function, that function will get executed by the Redux Thunk [middleware](#). This function doesn't need to be pure; it is thus allowed to have side effects, including executing asynchronous API calls. The function can also dispatch actions—like those synchronous actions we defined earlier.

We can still define these special thunk [action](#) creators inside our `actions.js` file:

`actions.js` (Asynchronous)

```

import fetch from 'cross-fetch'

```

```

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}

export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'
export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT,
    subreddit
  }
}

// Meet our first thunk action creator!
// Though its insides are different, you would use it just like any other action creator:
// store.dispatch(fetchPosts('reactjs'))

export function fetchPosts(subreddit) {
  // Thunk middleware knows how to handle functions.
  // It passes the dispatch method as an argument to the function,
  // thus making it able to dispatch actions itself.

  return function(dispatch) {
    // First dispatch: the app state is updated to inform
    // that the API call is starting.

    dispatch(requestPosts(subreddit))

    // The function called by the thunk middleware can return a value,
    // that is passed on as the return value of the dispatch method.

    // In this case, we return a promise to wait for.
    // This is not required by thunk middleware, but it is convenient for us.

    return fetch(`https://www.reddit.com/r/${subreddit}.json`)
      .then(
        response => response.json(),
        // Do not use catch, because that will also catch
        // any errors in the dispatch and resulting render,
        // causing a loop of 'Unexpected batch number' errors.
        // https://github.com/facebook/react/issues/6895
        error => console.log('An error occurred.', error)
      )
      .then(json =>
        // We can dispatch many times!
        // Here, we update the app state with the results of the API call.

        dispatch(receivePosts(subreddit, json))
      )
  }
}

```

Note on `fetch`

We use `fetch` API in the examples. It is a new API for making network requests that replaces `XMLHttpRequest` for most common needs. Because most browsers don't yet support it natively, we suggest that you use `cross-fetch` library:

```
// Do this in every file where you use `fetch`
import fetch from 'cross-fetch'
```

Internally, it uses `whatwg-fetch` polyfill on the client, and `node-fetch` on the server, so you won't need to change API calls if you change your app to be `universal`.

Be aware that any `fetch` polyfill assumes a `Promise` polyfill is already present. The easiest way to ensure you have a `Promise` polyfill is to enable Babel's ES6 polyfill in your entry point before any other code runs:

```
// Do this once before any other code in your app
import 'babel-polyfill'
```

How do we include the Redux Thunk `middleware` in the dispatch mechanism? We use the `applyMiddleware()` `store` enhancer from Redux, as shown below:

`index.js`

```
import thunkMiddleware from 'redux-thunk'
import { createLogger } from 'redux-logger'
import { createStore, applyMiddleware } from 'redux'
import { selectSubreddit, fetchPosts } from './actions'
import rootReducer from './reducers'

const loggerMiddleware = createLogger()

const store = createStore(
  rootReducer,
  applyMiddleware(
    thunkMiddleware, // lets us dispatch() functions
    loggerMiddleware // neat middleware that logs actions
  )
)

store.dispatch(selectSubreddit('reactjs'))
store.dispatch(fetchPosts('reactjs')).then(() => console.log(store.getState()))
```

The nice thing about thunks is that they can dispatch results of each other:

`actions.js` (with `fetch`)

```
import fetch from 'cross-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
```

```

function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}

export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'
export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT,
    subreddit
  }
}

function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`https://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
  }
}

function shouldFetchPosts(state, subreddit) {
  const posts = state.postsBySubreddit[subreddit]
  if (!posts) {
    return true
  } else if (posts.isFetching) {
    return false
  } else {
    return posts.didInvalidate
  }
}

export function fetchPostsIfNeeded(subreddit) {
  // Note that the function also receives getState()
  // which lets you choose what to dispatch next.

  // This is useful for avoiding a network request if
  // a cached value is already available.

  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), subreddit)) {
      // Dispatch a thunk from thunk!
      return dispatch(fetchPosts(subreddit))
    } else {
      // Let the calling code know there's nothing to wait for.
      return Promise.resolve()
    }
  }
}

```

This lets us write more sophisticated async control flow gradually, while the consuming code can stay pretty much the same:

index.js

```

store
  .dispatch(fetchPostsIfNeeded('reactjs'))
  .then(() => console.log(store.getState()))

```

Note about Server Rendering

Async [action](#) creators are especially convenient for server rendering. You can create a [store](#), dispatch a single async [action](#) creator that dispatches other async [action](#) creators to fetch data for a whole section of your app, and only render after the Promise it returns, completes. Then your [store](#) will already be hydrated with the [state](#) you need before rendering.

[Thunk middleware](#) isn't the only way to orchestrate asynchronous actions in Redux:

- You can use [redux-promise](#) or [redux-promise-middleware](#) to dispatch Promises instead of functions.
- You can use [redux-observable](#) to dispatch Observables.
- You can use the [redux-saga middleware](#) to build more complex asynchronous actions.
- You can use the [redux-pack middleware](#) to dispatch promise-based asynchronous actions.
- You can even write a custom [middleware](#) to describe calls to your API, like the [real world example](#) does.

It is up to you to try a few options, choose a convention you like, and follow it, whether with, or without the [middleware](#).

Connecting to UI

Dispatching async actions is no different from dispatching synchronous actions, so we won't discuss this in detail. See [Usage with React](#) for an introduction into using Redux from React components. See [Example: Reddit API](#) for the complete source code discussed in this example.

Next Steps

Read [Async Flow](#) to recap how async actions fit into the Redux flow.

Async Flow

Without [middleware](#), Redux [store](#) only supports [synchronous data flow](#). This is what you get by default with `createStore()`.

You may enhance `createStore()` with `applyMiddleware()`. It is not required, but it lets you [express asynchronous actions in a convenient way](#).

Asynchronous [middleware](#) like [redux-thunk](#) or [redux-promise](#) wraps the [store's](#) `dispatch()` method and allows you to dispatch something other than actions, for example, functions or Promises. Any [middleware](#) you use can then intercept anything you dispatch, and in turn, can pass actions to the next [middleware](#) in the chain. For example, a Promise [middleware](#) can intercept Promises and dispatch a pair of begin/end actions asynchronously in response to each Promise.

When the last [middleware](#) in the chain dispatches an [action](#), it has to be a plain object. This is when the [synchronous Redux data flow](#) takes place.

Check out [the full source code for the async example](#).

Next Steps

Now that you've seen an example of what [middleware](#) can do in Redux, it's time to learn how it actually works, and how you can create your own. Go on to the next detailed section about [Middleware](#).

Middleware

You've seen [middleware](#) in [action](#) in the [Async Actions](#) example. If you've used server-side libraries like [Express](#) and [Koa](#), you were also probably already familiar with the concept of [middleware](#). In these frameworks, [middleware](#) is some code you can put between the framework receiving a request, and the framework generating a response. For example, Express or Koa [middleware](#) may add CORS headers, logging, compression, and more. The best feature of [middleware](#) is that it's composable in a chain. You can use multiple independent third-party [middleware](#) in a single project.

Redux [middleware](#) solves different problems than Express or Koa [middleware](#), but in a conceptually similar way. **It provides a third-party extension point between dispatching an [action](#), and the moment it reaches the [reducer](#).** People use Redux [middleware](#) for logging, crash reporting, talking to an asynchronous API, routing, and more.

This article is divided into an in-depth intro to help you grok the concept, and [a few practical examples](#) to show the power of [middleware](#) at the very end. You may find it helpful to switch back and forth between them, as you flip between feeling bored and inspired.

Understanding Middleware

While [middleware](#) can be used for a variety of things, including asynchronous API calls, it's really important that you understand where it comes from. We'll guide you through the thought process leading to [middleware](#), by using logging and crash reporting as examples.

Problem: Logging

One of the benefits of Redux is that it makes [state](#) changes predictable and transparent. Every time an [action](#) is dispatched, the new [state](#) is computed and saved. The [state](#) cannot change by itself, it can only change as a consequence of a specific [action](#).

Wouldn't it be nice if we logged every [action](#) that happens in the app, together with the [state](#) computed after it? When something goes wrong, we can look back at our log, and figure out which [action](#) corrupted the [state](#).

▼ ADD_TODO
<div><div>❏</div> dispatching: <i>Object {type: "ADD_TODO", text: "Use Redux"}</i></div> <div>next state: ► <i>Object {visibilityFilter: "SHOW_ALL", todos: Array[1]}</i></div>
▼ ADD_TODO
<div><div>❏</div> dispatching: <i>Object {type: "ADD_TODO", text: "Learn about middleware"}</i></div> <div>next state: ► <i>Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}</i></div>
▼ COMPLETE_TODO
<div><div>❏</div> dispatching: <i>Object {type: "COMPLETE_TODO", index: 0}</i></div> <div>next state: ► <i>Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}</i></div>
▼ SET_VISIBILITY_FILTER
<div><div>❏</div> dispatching: <i>Object {type: "SET_VISIBILITY_FILTER", filter: "SHOW_COMPLETED"}</i></div> <div>next state: ► <i>Object {visibilityFilter: "SHOW_COMPLETED", todos: Array[2]}</i></div>

How do we approach this with Redux?

Attempt #1: Logging Manually

The most naïve solution is just to log the `action` and the next `state` yourself every time you call `store.dispatch(action)`. It's not really a solution, but just a first step towards understanding the problem.

Note

If you're using `react-redux` or similar bindings, you likely won't have direct access to the `store` instance in your components. For the next few paragraphs, just assume you pass the `store` down explicitly.

Say, you call this when creating a todo:

```
store.dispatch(addTodo('Use Redux'))
```

To log the `action` and `state`, you can change it to something like this:

```
const action = addTodo('Use Redux')

console.log('dispatching', action)
store.dispatch(action)
console.log('next state', store.getState())
```

This produces the desired effect, but you wouldn't want to do it every time.

Attempt #2: Wrapping Dispatch

You can extract logging into a function:

```
function dispatchAndLog(store, action) {
  console.log('dispatching', action)
  store.dispatch(action)
  console.log('next state', store.getState())
}
```

You can then use it everywhere instead of `store.dispatch()`:

```
dispatchAndLog(store, addTodo('Use Redux'))
```

We could end this here, but it's not very convenient to import a special function every time.

Attempt #3: Monkeypatching Dispatch

What if we just replace the `dispatch` function on the `store` instance? The Redux `store` is just a plain object with a few methods, and we're writing JavaScript, so we can just monkeypatch the `dispatch` implementation:

```
const next = store.dispatch
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}
```


This is already closer to what we want! No matter where we dispatch an [action](#), it is guaranteed to be logged. Monkeypatching never feels right, but we can live with this for now.

Problem: Crash Reporting

What if we want to apply **more than one** such transformation to `dispatch` ?

A different useful transformation that comes to my mind is reporting JavaScript errors in production. The global `window.onerror` event is not reliable because it doesn't provide stack information in some older browsers, which is crucial to understand why an error is happening.

Wouldn't it be useful if, any time an error is thrown as a result of dispatching an [action](#), we would send it to a crash reporting service like [Sentry](#) with the stack trace, the [action](#) that caused the error, and the current [state](#)? This way it's much easier to reproduce the error in development.

However, it is important that we keep logging and crash reporting separate. Ideally we want them to be different modules, potentially in different packages. Otherwise we can't have an ecosystem of such utilities. (Hint: we're slowly getting to what [middleware](#) is!)

If logging and crash reporting are separate utilities, they might look like this:

```
function patchStoreToAddLogging(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}

function patchStoreToAddCrashReporting(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndReportErrors(action) {
    try {
      return next(action)
    } catch (err) {
      console.error('Caught an exception!', err)
      Raven.captureException(err, {
        extra: {
          action,
          state: store.getState()
        }
      })
      throw err
    }
  }
}
```

If these functions are published as separate modules, we can later use them to patch our [store](#):

```
patchStoreToAddLogging(store)
patchStoreToAddCrashReporting(store)
```

Still, this isn't nice.

Attempt #4: Hiding Monkeypatching

Monkeypatching is a hack. “Replace any method you like”, what kind of API is that? Let's figure out the essence of it instead. Previously, our functions replaced `store.dispatch`. What if they *returned* the new `dispatch` function instead?

```
function logger(store) {
  const next = store.dispatch

  // Previously:
  // store.dispatch = function dispatchAndLog(action) {

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

We could provide a helper inside Redux that would apply the actual monkeypatching as an implementation detail:

```
function applyMiddlewareByMonkeypatching(store, middlewares) {
  middlewares = middlewares.slice()
  middlewares.reverse()

  // Transform dispatch function with each middleware.
  middlewares.forEach(middleware => (store.dispatch = middleware(store)))
}
```

We could use it to apply multiple `middleware` like this:

```
applyMiddlewareByMonkeypatching(store, [logger, crashReporter])
```

However, it is still monkeypatching.

The fact that we hide it inside the library doesn't alter this fact.

Attempt #5: Removing Monkeypatching

Why do we even overwrite `dispatch`? Of course, to be able to call it later, but there's also another reason: so that every `middleware` can access (and call) the previously wrapped `store.dispatch`:

```
function logger(store) {
  // Must point to the function returned by the previous middleware:
  const next = store.dispatch

  return function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}
```

It is essential to chaining `middleware`!

If `applyMiddlewareByMonkeypatching` doesn't assign `store.dispatch` immediately after processing the first `middleware`, `store.dispatch` will keep pointing to the original `dispatch` function. Then the second `middleware` will also be bound to the original `dispatch` function.

But there's also a different way to enable chaining. The `middleware` could accept the `next()` dispatch function as a parameter instead of reading it from the `store` instance.

```
function logger(store) {
  return function wrapDispatchToAddLogging(next) {
    return function dispatchAndLog(action) {
      console.log('dispatching', action)
      let result = next(action)
      console.log('next state', store.getState())
      return result
    }
  }
}
```

It's a “we need to go deeper” kind of moment, so it might take a while for this to make sense. The function cascade feels intimidating. ES6 arrow functions make this `currying` easier on eyes:

```
const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

This is exactly what Redux `middleware` looks like.

Now `middleware` takes the `next()` dispatch function, and returns a dispatch function, which in turn serves as `next()` to the `middleware` to the left, and so on. It's still useful to have access to some `store` methods like `getState()`, so `store` stays available as the top-level argument.

Attempt #6: Naïvely Applying the Middleware

Instead of `applyMiddlewareByMonkeypatching()`, we could write `applyMiddleware()` that first obtains the final, fully wrapped `dispatch()` function, and returns a copy of the `store` using it:

```
// Warning: Naïve implementation!
// That's *not* Redux API.
function applyMiddleware(store, middlewares) {
  middlewares = middlewares.slice()
```

```

middlewares.reverse()
let dispatch = store.dispatch
middlewares.forEach(middleware => (dispatch = middleware(store)(dispatch)))
return Object.assign({}, store, { dispatch })
}

```

The implementation of `applyMiddleware()` that ships with Redux is similar, but **different in three important aspects**:

- It only exposes a subset of the `store API` to the `middleware`: `dispatch(action)` and `getState()` .
- It does a bit of trickery to make sure that if you call `store.dispatch(action)` from your `middleware` instead of `next(action)` , the `action` will actually travel the whole `middleware` chain again, including the current `middleware`. This is useful for asynchronous `middleware`, as we have seen [previously](#). There is one caveat when calling `dispatch` during setup, described below.
- To ensure that you may only apply `middleware` once, it operates on `createStore()` rather than on `store` itself. Instead of `(store, middlewares) => store` , its signature is `(...middlewares) => (createStore) => createStore` .

Because it is cumbersome to apply functions to `createStore()` before using it, `createStore()` accepts an optional last argument to specify such functions.

Caveat: Dispatching During Setup

While `applyMiddleware` executes and sets up your `middleware`, the `store.dispatch` function will point to the vanilla version provided by `createStore` . Dispatching would result in no other `middleware` being applied. If you are expecting an interaction with another `middleware` during setup, you will probably be disappointed. Because of this unexpected behavior, `applyMiddleware` will throw an error if you try to dispatch an `action` before the set up completes. Instead, you should either communicate directly with that other `middleware` via a common object (for an API-calling `middleware`, this may be your API client object) or waiting until after the `middleware` is constructed with a callback.

The Final Approach

Given this `middleware` we just wrote:

```

const logger = store => next => action => {
  console.log('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  return result
}

const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}

```

Here's how to apply it to a Redux [store](#):

```
import { createStore, combineReducers, applyMiddleware } from 'redux'

const todoApp = combineReducers(reducers)
const store = createStore(
  todoApp,
  // applyMiddleware() tells createStore() how to handle middleware
  applyMiddleware(logger, crashReporter)
)
```

That's it! Now any actions dispatched to the [store](#) instance will flow through `logger` and `crashReporter` :

```
// Will flow through both logger and crashReporter middleware!
store.dispatch(addTodo('Use Redux'))
```

Seven Examples

If your head boiled from reading the above section, imagine what it was like to write it. This section is meant to be a relaxation for you and me, and will help get your gears turning.

Each function below is a valid Redux [middleware](#). They are not equally useful, but at least they are equally fun.

```
/**
 * Logs all actions and states after they are dispatched.
 */
const logger = store => next => action => {
  console.group(action.type)
  console.info('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  console.groupEnd()
  return result
}

/**
 * Sends crash reports as state is updated and listeners are notified.
 */
const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}

/**
 * Schedules actions with { meta: { delay: N } } to be delayed by N milliseconds.
 * Makes `dispatch` return a function to cancel the timeout in this case.
 */
const timeoutScheduler = store => next => action => {
  if (!action.meta || !action.meta.delay) {
    return next(action)
  }
}
```

```

    }

    const timeoutId = setTimeout(() => next(action), action.meta.delay)

    return function cancel() {
      clearTimeout(timeoutId)
    }
  }
}

/**
 * Schedules actions with { meta: { raf: true } } to be dispatched inside a rAF loop
 * frame. Makes `dispatch` return a function to remove the action from the queue in
 * this case.
 */
const rafScheduler = store => next => {
  const queuedActions = []
  let frame = null

  function loop() {
    frame = null
    try {
      if (queuedActions.length) {
        next(queuedActions.shift())
      }
    } finally {
      maybeRaf()
    }
  }

  function maybeRaf() {
    if (queuedActions.length && !frame) {
      frame = requestAnimationFrame(loop)
    }
  }

  return action => {
    if (!action.meta || !action.meta.raf) {
      return next(action)
    }

    queuedActions.push(action)
    maybeRaf()

    return function cancel() {
      queuedActions = queuedActions.filter(a => a !== action)
    }
  }
}

/**
 * Lets you dispatch promises in addition to actions.
 * If the promise is resolved, its result will be dispatched as an action.
 * The promise is returned from `dispatch` so the caller may handle rejection.
 */
const vanillaPromise = store => next => action => {
  if (typeof action.then !== 'function') {
    return next(action)
  }

  return Promise.resolve(action).then(store.dispatch)
}

/**
 * Lets you dispatch special actions with a { promise } field.
 *
 * This middleware will turn them into a single action at the beginning,

```

```

* and a single success (or failure) action when the `promise` resolves.
*
* For convenience, `dispatch` will return the promise so the caller can wait.
*/
const readyStatePromise = store => next => action => {
  if (!action.promise) {
    return next(action)
  }

  function makeAction(ready, data) {
    const newAction = Object.assign({}, action, { ready }, data)
    delete newAction.promise
    return newAction
  }

  next(makeAction(false))
  return action.promise.then(
    result => next(makeAction(true, { result })),
    error => next(makeAction(true, { error }))
  )
}

/**
* Lets you dispatch a function instead of an action.
* This function will receive `dispatch` and `getState` as arguments.
*
* Useful for early exits (conditions over `getState()`), as well
* as for async control flow (it can `dispatch()` something else).
*
* `dispatch` will return the return value of the dispatched function.
*/
const thunk = store => next => action =>
  typeof action === 'function'
    ? action(store.dispatch, store.getState)
    : next(action)

// You can use all of them! (It doesn't mean you should.)
const todoApp = combineReducers(reducers)
const store = createStore(
  todoApp,
  applyMiddleware(
    rafScheduler,
    timeoutScheduler,
    thunk,
    vanillaPromise,
    readyStatePromise,
    logger,
    crashReporter
  )
)

```

Usage with React Router

So you want to do routing with your Redux app. You can use it with [React Router](#). Redux will be the source of truth for your data and React Router will be the source of truth for your URL. In most of the cases, **it is fine** to have them separate unless you need to time travel and rewind actions that trigger a URL change.

Installing React Router

`react-router-dom` is available on npm . This guides assumes you are using `react-router-dom@^4.1.1` .

```
npm install --save react-router-dom
```

Configuring the Fallback URL

Before integrating React Router, we need to configure our development server. Indeed, our development server may be unaware of the declared routes in React Router configuration. For example, if you access `/todos` and refresh, your development server needs to be instructed to serve `index.html` because it is a single-page app. Here's how to enable this with popular development servers.

Note on Create React App

If you are using Create React App, you won't need to configure a fallback URL, it is automatically done.

Configuring Express

If you are serving your `index.html` from Express:

```
app.get('/*', (req, res) => {  
  res.sendFile(path.join(__dirname, 'index.html'))  
})
```

Configuring WebpackDevServer

If you are serving your `index.html` from WebpackDevServer: You can add to your `webpack.config.dev.js`:

```
devServer: {  
  historyApiFallback: true  
}
```

Connecting React Router with Redux App

Along this chapter, we will be using the [Todos](#) example. We recommend you to clone it while reading this chapter.

First we will need to import `<Router />` and `<Route />` from React Router. Here's how to do it:

```
import { BrowserRouter as Router, Route } from 'react-router-dom'
```


In a React app, usually you would wrap `<Route />` in `<Router />` so that when the URL changes, `<Router />` will match a branch of its routes, and render their configured components. `<Route />` is used to declaratively map routes to your application's component hierarchy. You would declare in `path` the path used in the URL and in `component` the single component to be rendered when the route matches the URL.

```
const Root = () => (  
  <Router>  
    <Route path="/" component={App} />  
  </Router>  
)
```

However, in our Redux App we will still need `<Provider />`. `<Provider />` is the higher-order component provided by React Redux that lets you bind Redux to React (see [Usage with React](#)).

We will then import the `<Provider />` from React Redux:

```
import { Provider } from 'react-redux'
```

We will wrap `<Router />` in `<Provider />` so that route handlers can get [access to the store](#).

```
const Root = ({ store }) => (  
  <Provider store={store}>  
    <Router>  
      <Route path="/" component={App} />  
    </Router>  
  </Provider>  
)
```

Now the `<App />` component will be rendered if the URL matches '/'. Additionally, we will add the optional `:filter?` parameter to `/`, because we will need it further on when we try to read the parameter `:filter` from the URL.

```
<Route path="/:filter?" component={App} />
```

components/Root.js

```
import React from 'react'  
import PropTypes from 'prop-types'  
import { Provider } from 'react-redux'  
import { BrowserRouter as Router, Route } from 'react-router-dom'  
import App from './App'  
  
const Root = ({ store }) => (  
  <Provider store={store}>  
    <Router>  
      <Route path="/:filter?" component={App} />  
    </Router>  
  </Provider>  
)  
  
Root.propTypes = {  
  store: PropTypes.object.isRequired  
}  
  
export default Root
```

We will also need to refactor `index.js` to render the `<Root />` component to the DOM.

`index.js`

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import todoApp from './reducers'
import Root from './components/Root'

const store = createStore(todoApp)

render(<Root store={store} />, document.getElementById('root'))
```

Navigating with React Router

React Router comes with a `<Link />` component that lets you navigate around your application. If you want to add some styles, `react-router-dom` has another special `<Link />` called `<NavLink />`, which accepts styling props. For instance, the `activeStyle` property lets us apply a style on the active [state](#).

In our example, we can wrap `<NavLink />` with a new container component `<FilterLink />` so as to dynamically change the URL.

`containers/FilterLink.js`

```
import React from 'react'
import { NavLink } from 'react-router-dom'

const FilterLink = ({ filter, children }) => (
  <NavLink
    exact
    to={filter === 'SHOW_ALL' ? '/' : `/${filter}`}
    activeStyle={{
      textDecoration: 'none',
      color: 'black'
    }}
  >
    {children}
  </NavLink>
)

export default FilterLink
```

`components/Footer.js`

```
import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {' '}
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {' '}
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)
```

```

    </p>
  )

  export default Footer

```

Now if you click on `<FilterLink />` you will see that your URL will change between `'/SHOW_COMPLETED'`, `'/SHOW_ACTIVE'`, and `'/'`. Even if you are going back with your browser, it will use your browser's history and effectively go to your previous URL.

Reading From the URL

Currently, the todo list is not filtered even after the URL changed. This is because we are filtering from `<VisibleTodoList />`'s `mapStateToProps()`, which is still bound to the `state` and not to the URL. `mapStateToProps` has an optional second argument `ownProps` that is an object with every props passed to `<VisibleTodoList />`.

containers/VisibleTodoList.js

```

const mapStateToProps = (state, ownProps) => {
  return {
    todos: getVisibleTodos(state.todos, ownProps.filter) // previously was getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

```

Right now we are not passing anything to `<App />` so `ownProps` is an empty object. To filter our todos according to the URL, we want to pass the URL params to `<VisibleTodoList />`.

When previously we wrote: `<Route path="/:filter?" component={App} />`, it made available inside `App` a `params` property.

`params` property is an object with every param specified in the url with the `match` object. e.g: `match.params` will be equal to `{ filter: 'SHOW_COMPLETED' }` if we are navigating to `localhost:3000/SHOW_COMPLETED`. We can now read the URL from `<App />`.

Note that we are using [ES6 destructuring](#) on the properties to pass in `params` to `<VisibleTodoList />`.

components/App.js

```

const App = ({ match: { params } }) => {
  return (
    <div>
      <AddTodo />
      <VisibleTodoList filter={params.filter || 'SHOW_ALL'} />
      <Footer />
    </div>
  )
}

```

Next Steps

Now that you know how to do basic routing, you can learn more about [React Router API](#)

Note About Other Routing Libraries

Redux Router is an experimental library, it lets you keep entirely the [state](#) of your URL inside your [redux store](#). It has the same API with React Router API but has a smaller community support than react-router.

React Router Redux creates a binding between your redux app and react-router and it keeps them in sync. Without this binding, you will not be able to rewind the actions with Time Travel. Unless you need this, React Router and Redux can operate completely apart.

Example: Reddit API

This is the complete source code of the Reddit headline fetching example we built during the [advanced tutorial](#).

Entry Point

index.js

```
import 'babel-polyfill'

import React from 'react'
import { render } from 'react-dom'
import Root from '../containers/Root'

render(<Root />, document.getElementById('root'))
```

Action Creators and Constants

actions.js

```
import fetch from 'cross-fetch'

export const REQUEST_POSTS = 'REQUEST_POSTS'
export const RECEIVE_POSTS = 'RECEIVE_POSTS'
export const SELECT_SUBREDDIT = 'SELECT_SUBREDDIT'
export const INVALIDATE_SUBREDDIT = 'INVALIDATE_SUBREDDIT'

export function selectSubreddit(subreddit) {
  return {
    type: SELECT_SUBREDDIT,
    subreddit
  }
}

export function invalidateSubreddit(subreddit) {
  return {
    type: INVALIDATE_SUBREDDIT,
    subreddit
  }
}

function requestPosts(subreddit) {
  return {
    type: REQUEST_POSTS,
    subreddit
  }
}

function receivePosts(subreddit, json) {
  return {
    type: RECEIVE_POSTS,
    subreddit,
    posts: json.data.children.map(child => child.data),
    receivedAt: Date.now()
  }
}
```

```

}

function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`https://www.reddit.com/r/${subreddit}.json`)
      .then(response => response.json())
      .then(json => dispatch(receivePosts(subreddit, json)))
  }
}

function shouldFetchPosts(state, subreddit) {
  const posts = state.postsBySubreddit[subreddit]
  if (!posts) {
    return true
  } else if (posts.isFetching) {
    return false
  } else {
    return posts.didInvalidate
  }
}

export function fetchPostsIfNeeded(subreddit) {
  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), subreddit)) {
      return dispatch(fetchPosts(subreddit))
    }
  }
}

```

Reducers

reducers.js

```

import { combineReducers } from 'redux'
import {
  SELECT_SUBREDDIT,
  INVALIDATE_SUBREDDIT,
  REQUEST_POSTS,
  RECEIVE_POSTS
} from './actions'

function selectedSubreddit(state = 'reactjs', action) {
  switch (action.type) {
    case SELECT_SUBREDDIT:
      return action.subreddit
    default:
      return state
  }
}

function posts(
  state = {
    isFetching: false,
    didInvalidate: false,
    items: []
  },
  action
) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
      return Object.assign({}, state, {

```

```

        didInvalidate: true
      })
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        isFetching: true,
        didInvalidate: false
      })
    case RECEIVE_POSTS:
      return Object.assign({}, state, {
        isFetching: false,
        didInvalidate: false,
        items: action.posts,
        lastUpdated: action.receivedAt
      })
    default:
      return state
  }
}

function postsBySubreddit(state = {}, action) {
  switch (action.type) {
    case INVALIDATE_SUBREDDIT:
    case RECEIVE_POSTS:
    case REQUEST_POSTS:
      return Object.assign({}, state, {
        [action.subreddit]: posts(state[action.subreddit], action)
      })
    default:
      return state
  }
}

const rootReducer = combineReducers({
  postsBySubreddit,
  selectedSubreddit
})

export default rootReducer

```

Store

configureStore.js

```

import { createStore, applyMiddleware } from 'redux'
import thunkMiddleware from 'redux-thunk'
import { createLogger } from 'redux-logger'
import rootReducer from './reducers'

const loggerMiddleware = createLogger()

export default function configureStore(preloadedState) {
  return createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(thunkMiddleware, loggerMiddleware)
  )
}

```

Container Components

containers/Root.js

```
import React, { Component } from 'react'
import { Provider } from 'react-redux'
import configureStore from '../configureStore'
import AsyncApp from './AsyncApp'

const store = configureStore()

export default class Root extends Component {
  render() {
    return (
      <Provider store={store}>
        <AsyncApp />
      </Provider>
    )
  }
}
```

containers/AsyncApp.js

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import {
  selectSubreddit,
  fetchPostsIfNeeded,
  invalidateSubreddit
} from '../actions'
import Picker from '../components/Picker'
import Posts from '../components/Posts'

class AsyncApp extends Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.handleRefreshClick = this.handleRefreshClick.bind(this)
  }

  componentDidMount() {
    const { dispatch, selectedSubreddit } = this.props
    dispatch(fetchPostsIfNeeded(selectedSubreddit))
  }

  componentDidUpdate(prevProps) {
    if (this.props.selectedSubreddit !== prevProps.selectedSubreddit) {
      const { dispatch, selectedSubreddit } = this.props
      dispatch(fetchPostsIfNeeded(selectedSubreddit))
    }
  }

  handleChange(nextSubreddit) {
    this.props.dispatch(selectSubreddit(nextSubreddit))
    this.props.dispatch(fetchPostsIfNeeded(nextSubreddit))
  }

  handleRefreshClick(e) {
    e.preventDefault()

    const { dispatch, selectedSubreddit } = this.props
    dispatch(invalidateSubreddit(selectedSubreddit))
    dispatch(fetchPostsIfNeeded(selectedSubreddit))
  }
}
```



```

render() {
  const { selectedSubreddit, posts, isFetching, lastUpdated } = this.props
  return (
    <div>
      <Picker
        value={selectedSubreddit}
        onChange={this.handleChange}
        options={['reactjs', 'frontend']}
      />
      <p>
        {lastUpdated && (
          <span>
            Last updated at {new Date(lastUpdated).toLocaleTimeString()}.{' '}
          </span>
        )}
        {!isFetching && (
          <button onClick={this.handleRefreshClick}>Refresh</button>
        )}
      </p>
      {isFetching && posts.length === 0 && <h2>Loading...</h2>}
      {!isFetching && posts.length === 0 && <h2>Empty.</h2>}
      {posts.length > 0 && (
        <div style={{ opacity: isFetching ? 0.5 : 1 }}>
          <Posts posts={posts} />
        </div>
      )}
    </div>
  )
}
}

AsyncApp.propTypes = {
  selectedSubreddit: PropTypes.string.isRequired,
  posts: PropTypes.array.isRequired,
  isFetching: PropTypes.bool.isRequired,
  lastUpdated: PropTypes.number,
  dispatch: PropTypes.func.isRequired
}

function mapStateToProps(state) {
  const { selectedSubreddit, postsBySubreddit } = state
  const { isFetching, lastUpdated, items: posts } = postsBySubreddit[
    selectedSubreddit
  ] || {
    isFetching: true,
    items: []
  }

  return {
    selectedSubreddit,
    posts,
    isFetching,
    lastUpdated
  }
}

export default connect(mapStateToProps)(AsyncApp)

```

Presentational Components

components/Picker.js

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'

export default class Picker extends Component {
  render() {
    const { value, onChange, options } = this.props

    return (
      <span>
        <h1>{value}</h1>
        <select onChange={e => onChange(e.target.value)} value={value}>
          {options.map(option => (
            <option value={option} key={option}>
              {option}
            </option>
          ))}
        </select>
      </span>
    )
  }
}

Picker.propTypes = {
  options: PropTypes.arrayOf(PropTypes.string.isRequired).isRequired,
  value: PropTypes.string.isRequired,
  onChange: PropTypes.func.isRequired
}

```

components/Posts.js

```

import React, { Component } from 'react'
import PropTypes from 'prop-types'

export default class Posts extends Component {
  render() {
    return (
      <ul>
        {this.props.posts.map((post, i) => (
          <li key={i}>{post.title}</li>
        ))}
      </ul>
    )
  }
}

Posts.propTypes = {
  posts: PropTypes.array.isRequired
}

```

Next Steps

If you landed in this section, you might be wondering at this point, "what should I do now?". Here is where we provide some essential tips/suggestions on how to diverge from creating trivial TodoMVC apps to a real world application.

Tips & Considerations For The Real World

Whenever we decide to create a new project, we tend to bypass several aspects that in the future may slow us down. In a real world project we have to consider several things before we start coding, such as: how to configure a `store`, `store` size, data structure, `state` model, middlewares, environment, async transactions, immutability, etc..

The above are some of the main considerations we have to think about beforehand. It's not an easy task, but there are some strategies for making it go smoothly.

UI vs State

One of the biggest challenges developers face when using Redux is to *describe UI `state` with data*. The majority of software programs out there are just data transformation, and having the clear understanding that UIs are simply data beautifully presented facilitates the process of building them.

Nicolas Hery describes it really well in "*Describing UI `state` with data*". Also, it's always good to know *When to use Redux*, because a lot of times *You Might Not Need Redux*

Configure a Store

To configure a `store` we have to make major considerations on which `middleware` to use. There are several libraries out there, but the most popular ones are:

Perform Asynchronous dispatch

- `redux-thunk`
 - Redux Thunk `middleware` allows you to write `action` creators that return a function instead of an `action`. The thunk can be used to delay the dispatch of an `action`, or to dispatch only if a certain condition is met. It incorporates the methods `dispatch` and `getState` as parameters.
- `redux-saga`
 - `redux-saga` is a library that aims to make the execution of application side effects (e.g., asynchronous tasks like data fetching and impure procedures such as accessing the browser cache) manageable and efficient. It's simple to test, as it uses the ES6 feature called `generators`, making the flow as easy to read as synchronous code.
- `redux-observable`
 - `redux-observable` is a `middleware` for `redux` that is inspired by `redux-thunk`. It allows developers to dispatch a function that returns an `Observable`, `Promise` or `iterable` of `action(s)`. When the observable emits an `action`, or the promise resolves an `action`, or the iterable gives an `action` out, that `action` is then dispatched as usual.

Development Purposes / debug

- [redux-devtools](#)
 - Redux DevTools is a set of tools for your Redux development workflow.
- [redux-logger](#)
 - redux-logger logs all actions that are being dispatched to the [store](#).

To be able to choose one of these libraries we must take into account whether we are building a small or large application. Usability, code standards, and JavaScript knowledge may also be considered. All of them are similar.

Tip: Think of middlewares as **skills** you give to your `store`. i.e: By attributing the `redux-thunk` to your [store](#), you're giving the `store` the ability to dispatch async actions.

Naming Convention

A big source of confusion when it comes to a large project is what to name things. This is often just as important as the code itself. Defining a naming convention for your actions at the very beginning of a project and sticking to that convention helps you to scale up as the scope of the project grows.

Great source: [A Simple Naming Convention for Action Creators in Redux](#) and [Redux Patterns and Anti-Patterns](#)

Tip: Set up an opinionated code formatter, such as [Prettier](#).

Scalability

There is no magic to analyze and predict how much your application is going to grow. But it's okay! Redux's simplistic foundation means it will adapt to many kinds of applications as they grow. Here are some resources on how to build up your application in a sensible manner:

- [Taming Large React Applications with Redux](#)
- [Real-World React and Redux - part I](#)
- [Real-World React and Redux - part II](#)
- [Redux: Architecting and scaling a new web app at the NY Times](#)

Tip: It's great to plan things beforehand, but don't get caught up in "analysis paralysis". Done is always better than perfect, after all. And [Redux makes refactoring easy](#) if you need to.

With all that being said, the best practice is to keep coding and learning. Participate in [issues](#) and [StackOverflow questions](#). Helping others is a great way of mastering Redux.

Tip: A repository with an extensive amount of content about best practices and Redux architecture is shared by [@markerikson](#) at [react-redux-links](#).

Recipes

These are some use cases and code snippets to get you started with Redux in a real app. They assume you understand the topics in [basic](#) and [advanced](#) tutorials.

- [Configuring Your Store](#)
- [Usage with TypeScript](#)
- [Migrating to Redux](#)
- [Using Object Spread Operator](#)
- [Reducing Boilerplate](#)
- [Code Splitting](#)
- [Server Rendering](#)
- [Writing Tests](#)
- [Computing Derived Data](#)
- [Implementing Undo History](#)
- [Isolating Subapps](#)
- [Structuring Reducers](#)
- [Using Immutable.JS with Redux](#)

Configuring Your Store

In the [basics section](#), we introduced the fundamental Redux concepts by building an example Todo list app.

We will now explore how to customise the [store](#) to add extra functionality. We'll start with the source code from the basics section, which you can view in [the documentation](#), in [our repository of examples](#), or [in your browser via CodeSandbox](#).

Creating the store

First, let's look at the original `index.js` file in which we created our [store](#):

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import rootReducer from './reducers'
import App from './components/App'

const store = createStore(rootReducer)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

In this code, we pass our reducers to the Redux `createStore` function, which returns a `store` object. We then pass this object to the `react-redux` `Provider` component, which is rendered at the top of our component tree.

This ensures that any time we connect to Redux in our app via `react-redux` `connect`, the [store](#) is available to our components.

Extending Redux functionality

Most apps extend the functionality of their Redux [store](#) by adding [middleware](#) or [store](#) enhancers (*note: [middleware](#) is common, enhancers are less common*). [Middleware](#) adds extra functionality to the Redux `dispatch` function; enhancers add extra functionality to the Redux [store](#).

We will add two middlewares and one enhancer:

- The `redux-thunk` [middleware](#), which allows simple asynchronous use of dispatch.
- A [middleware](#) which logs dispatched actions and the resulting new [state](#).
- An enhancer which logs the time taken for the reducers to process each [action](#).

Install `redux-thunk`

```
npm install --save redux-thunk
```

middleware/logger.js

```
const logger = store => next => action => {
  console.group(action.type)
  console.info('dispatching', action)
  let result = next(action)
  console.log('next state', store.getState())
  console.groupEnd()
  return result
}

export default logger
```

enhancers/monitorReducer.js

```
const round = number => Math.round(number * 100) / 100

const monitorReducerEnhancer = createStore => (
  reducer,
  initialState,
  enhancer
) => {
  const monitoredReducer = (state, action) => {
    const start = performance.now()
    const newState = reducer(state, action)
    const end = performance.now()
    const diff = round(end - start)

    console.log('reducer process time:', diff)

    return newState
  }

  return createStore(monitoredReducer, initialState, enhancer)
}

export default monitorReducerEnhancer
```

Let's add these to our existing `index.js` .

- First, we need to import `redux-thunk` plus our `loggerMiddleware` and `monitorReducerEnhancer` , plus two extra functions provided by Redux: `applyMiddleware` and `compose` .
- We then use `applyMiddleware` to create a [store](#) enhancer which will apply our `loggerMiddleware` and the `thunkMiddleware` to the [store](#)'s dispatch function.
- Next, we use `compose` to compose our new `middlewareEnhancer` and our `monitorReducerEnhancer` into one function.

This is needed because you can only pass one enhancer into `createStore` . To use multiple enhancers, you must first compose them into a single larger enhancer, as shown in this example.

- Finally, we pass this new `composedEnhancers` function into `createStore` as its third argument. *Note: the second argument, which we will ignore, lets you preload [state](#) into the [store](#).*

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { applyMiddleware, createStore, compose } from 'redux'
import thunkMiddleware from 'redux-thunk'
```

```
import rootReducer from './reducers'
import loggerMiddleware from './middleware/logger'
import monitorReducerEnhancer from './enhancers/monitorReducer'
import App from './components/App'

const middlewareEnhancer = applyMiddleware(loggerMiddleware, thunkMiddleware)
const composedEnhancers = compose(
  middlewareEnhancer,
  monitorReducerEnhancer
)

const store = createStore(rootReducer, undefined, composedEnhancers)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Problems with this approach

While this code works, for a typical app it is not ideal.

Most apps use more than one [middleware](#), and each [middleware](#) often requires some initial setup. The extra noise added to the `index.js` can quickly make it hard to maintain, because the logic is not cleanly organised.

The solution: `configureStore`

The solution to this problem is to create a new `configureStore` function which encapsulates our [store](#) creation logic, which can then be located in its own file to ease extensibility.

The end goal is for our `index.js` to look like this:

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './components/App'
import configureStore from './configureStore'

const store = configureStore()

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

All the logic related to configuring the [store](#) - including importing reducers, [middleware](#), and enhancers - is handled in a dedicated file.

To achieve this, `configureStore` function looks like this:

```
import { applyMiddleware, compose, createStore } from 'redux'
import thunkMiddleware from 'redux-thunk'
```



```
import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureStore(preloadedState) {
  const middlewares = [loggerMiddleware, thunkMiddleware]
  const middlewareEnhancer = applyMiddleware(...middlewares)

  const enhancers = [middlewareEnhancer, monitorReducersEnhancer]
  const composedEnhancers = compose(...enhancers)

  const store = createStore(rootReducer, preloadedState, composedEnhancers)

  return store
}
```

This function follows the same steps outlined above, with some of the logic split out to prepare for extension, which will make it easier to add more in future:

- Both `middlewares` and `enhancers` are defined as arrays, separate from the functions which consume them.

This allows us to easily add more `middleware` or enhancers based on different conditions.

For example, it is common to add some `middleware` only when in development mode, which is easily achieved by pushing to the `middlewares` array inside an if statement:

```
if (process.env.NODE_ENV === 'development') {
  middlewares.push(secretMiddleware)
}
```

- A `preloadedState` variable is passed through to `createStore` in case we want to add this later.

This also makes our `createStore` function easier to reason about - each step is clearly separated, which makes it more obvious what exactly is happening.

Integrating the devtools extension

Another common feature which you may wish to add to your app is the `redux-devtools-extension` integration.

The extension is a suite of tools which give you absolute control over your Redux `store` - it allows you to inspect and replay actions, explore your `state` at different times, dispatch actions directly to the `store`, and much more. [Click here to read more about the available features.](#)

There are several ways to integrate the extension, but we will use the most convenient option.

First, we install the package via npm:

```
npm install --save-dev redux-devtools-extension
```

Next, we remove the `compose` function which we imported from `redux`, and replace it with a new `composeWithDevTools` function imported from `redux-devtools-extension`.

The final code looks like this:

```
import { applyMiddleware, createStore } from 'redux'
import thunkMiddleware from 'redux-thunk'
import { composeWithDevTools } from 'redux-devtools-extension'
```

```
import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureStore(preloadedState) {
  const middlewares = [loggerMiddleware, thunkMiddleware]
  const middlewareEnhancer = applyMiddleware(...middlewares)

  const enhancers = [middlewareEnhancer, monitorReducersEnhancer]
  const composedEnhancers = composeWithDevTools(...enhancers)

  const store = createStore(rootReducer, preloadedState, composedEnhancers)

  return store
}
```

And that's it!

If we now visit our app via a browser with the devtools extension installed, we can explore and debug using a powerful new tool.

Hot reloading

Another powerful tool which can make the development process a lot more intuitive is hot reloading, which means replacing pieces of code without restarting your whole app.

For example, consider what happens when you run your app, interact with it for a while, and then decide to make changes to one of your reducers. Normally, when you make those changes your app will restart, reverting your Redux [state](#) to its initial value.

With hot module reloading enabled, only the [reducer](#) you changed would be reloaded, allowing you to change your code *without* resetting the [state](#) every time. This makes for a much faster development process.

We'll add hot reloading both to our Redux reducers and to our React components.

First, let's add it to our `configureStore` function:

```
import { applyMiddleware, compose, createStore } from 'redux'
import thunkMiddleware from 'redux-thunk'

import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureStore(preloadedState) {
  const middlewares = [loggerMiddleware, thunkMiddleware]
  const middlewareEnhancer = applyMiddleware(...middlewares)

  const enhancers = [middlewareEnhancer, monitorReducersEnhancer]
  const composedEnhancers = compose(...enhancers)

  const store = createStore(rootReducer, preloadedState, composedEnhancers)

  if (process.env.NODE_ENV !== 'production' && module.hot) {
    module.hot.accept('./reducers', () => store.replaceReducer(rootReducer))
  }

  return store
}
```

The new code is wrapped in an `if` statement, so it only runs when our app is not in production mode, and only if the `module.hot` feature is available.

Bundlers like Webpack and Parcel support a `module.hot.accept` method to specify which module should be hot reloaded, and what should happen when the module changes. In this case, we're watching the `./reducers` module, and passing the updated `rootReducer` to the `store.replaceReducer` method when it changes.

We'll also use the same pattern in our `index.js` to hot reload any changes to our React components:

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './components/App'
import configureStore from './configureStore'

const store = configureStore()

const renderApp = () =>
  render(
    <Provider store={store}>
      <App />
    </Provider>,
    document.getElementById('root')
  )

if (process.env.NODE_ENV !== 'production' && module.hot) {
  module.hot.accept('./components/App', renderApp)
}

renderApp()
```

The only extra change here is that we have encapsulated our app's rendering into a new `renderApp` function, which we now call to re-render the app.

Simplifying Setup with Redux Starter Kit

The Redux core library is deliberately unopinionated. It lets you decide how you want to handle everything, like `store` setup, what your `state` contains, and how you want to build your reducers.

This is good in some cases, because it gives you flexibility, but that flexibility isn't always needed. Sometimes we just want the simplest possible way to get started, with some good default behavior out of the box.

The [Redux Starter Kit](#) package is designed to help simplify several common Redux use cases, including `store` setup. Let's see how it can help improve the `store` setup process.

Redux Starter Kit includes a prebuilt `configureStore` function like the one shown in the earlier examples.

The fastest way to use it is to just pass the root `reducer` function:

```
import { configureStore } from 'redux-starter-kit'
import rootReducer from './reducers'

const store = configureStore({
  reducer: rootReducer
})

export default store
```

Note that it accepts an object with named parameters, to make it clearer what you're passing in.

By default, `configureStore` from Redux Starter Kit will:

- Call `applyMiddleware` with a default list of middleware, including `redux-thunk`, and some development-only middleware that catch common mistakes like mutating state
- Call `composeWithDevTools` to set up the Redux DevTools Extension

Here's what the hot reloading example might look like using Redux Starter Kit:

```
import { configureStore, getDefaultMiddleware } from 'redux-starter-kit'

import monitorReducersEnhancer from './enhancers/monitorReducers'
import loggerMiddleware from './middleware/logger'
import rootReducer from './reducers'

export default function configureAppStore(preloadedState) {
  const store = configureStore({
    reducer: rootReducer,
    middleware: [loggerMiddleware, ...getDefaultMiddleware()],
    preloadedState,
    enhancers: [monitorReducersEnhancer]
  })

  if (process.env.NODE_ENV !== 'production' && module.hot) {
    module.hot.accept('./reducers', () => store.replaceReducer(rootReducer))
  }

  return store
}
```

That definitely simplifies some of the setup process.

Next Steps

Now that you know how to encapsulate your `store` configuration to make it easier to maintain, you can [learn more about the advanced features Redux provides](#), or take a closer look at some of the [extensions available in the Redux ecosystem](#).

Migrating to Redux

Redux is not a monolithic framework, but a set of contracts and a [few functions that make them work together](#). The majority of your “Redux code” will not even use Redux APIs, as most of the time you'll be writing functions.

This makes it easy to migrate both to and from Redux.
We don't want to lock you in!

From Flux

[Reducers](#) capture “the essence” of Flux Stores, so it's possible to gradually migrate an existing Flux project towards Redux, whether you are using [Flummox](#), [Alt](#), [traditional Flux](#), or any other Flux library.

Your process will look like this:

- Create a function called `createFluxStore(reducer)` that creates a Flux [store](#) compatible with your existing app from a [reducer](#) function. Internally it might look similar to `createStore` ([source](#)) implementation from Redux. Its dispatch handler should just call the `reducer` for any [action](#), [store](#) the next [state](#), and emit change.
- This allows you to gradually rewrite every Flux [Store](#) in your app as a [reducer](#), but still export `createFluxStore(reducer)` so the rest of your app is not aware that this is happening and sees the Flux stores.
- As you rewrite your Stores, you will find that you need to avoid certain Flux anti-patterns such as fetching API inside the [Store](#), or triggering actions inside the Stores. Your Flux code will be easier to follow once you port it to be based on reducers!
- When you have ported all of your Flux Stores to be implemented on top of reducers, you can replace the Flux library with a single Redux [store](#), and combine those reducers you already have into one using `combineReducers(reducers)`.
- Now all that's left to do is to port the UI to [use react-redux](#) or equivalent.
- Finally, you might want to begin using some Redux idioms like [middleware](#) to further simplify your asynchronous code.

From Backbone

Backbone's model layer is quite different from Redux, so we don't suggest mixing them. If possible, it is best that you rewrite your app's model layer from scratch instead of connecting Backbone to Redux. However, if a rewrite is not feasible, you may use [backbone-redux](#) to migrate gradually, and keep the Redux [store](#) in sync with Backbone models and collections.

If your Backbone codebase is too big for a quick rewrite or you don't want to manage interactions between [store](#) and models, use [backbone-redux-migrator](#) to help your two codebases coexist while keeping healthy separation. Once your rewrite finishes, Backbone code can be discarded and your Redux application can work on its own once you configure router.

Using Object Spread Operator

Since one of the core tenets of Redux is to never mutate `state`, you'll often find yourself using `Object.assign()` to create copies of objects with new or updated values. For example, in the `todoApp` below `Object.assign()` is used to return a new `state` object with an updated `visibilityFilter` property:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

While effective, using `Object.assign()` can quickly make simple reducers difficult to read given its rather verbose syntax.

An alternative approach is to use the `object spread syntax` recently added to the JavaScript specification. It lets you use the spread (`...`) operator to copy enumerable properties from one object to another in a more succinct way. The object spread operator is conceptually similar to the ES6 `array spread operator`. We can simplify the `todoApp` example above by using the object spread syntax:

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return { ...state, visibilityFilter: action.filter }
    default:
      return state
  }
}
```

The advantage of using the object spread syntax becomes more apparent when you're composing complex objects. Below `getAddedIds` maps an array of `id` values to an array of objects with values returned from `getProduct` and `getQuantity`.

```
return getAddedIds(state.cart).map(id =>
  Object.assign({}, getProduct(state.products, id), {
    quantity: getQuantity(state.cart, id)
  })
)
```

Object spread lets us simplify the above `map` call to:

```
return getAddedIds(state.cart).map(id => ({
  ...getProduct(state.products, id),
  quantity: getQuantity(state.cart, id)
}))
```

While the object spread syntax is a [Stage 4](#) proposal for ECMAScript and accepted for the 2018 specification release, you will still need to use a transpiler such as [Babel](#) to use it in production systems. You should use the [env](#) preset, install [@babel/plugin-proposal-object-rest-spread](#) and add it individually to the `plugins` array in your `.babelrc`.

```
{
  "presets": ["@babel/preset-env"],
  "plugins": ["@babel/plugin-proposal-object-rest-spread"]
}
```

Note on Object Spread Operator

Like the Array Spread Operator, the Object Spread Operator creates a [shallow clone](#) of the original object. In other words, for multidimensional source objects, elements in the copied object at a depth greater than one are mere references to the source object (with the exception of [primitives](#), which are copied). Thus, you cannot reliably use the Object Spread Operator (`...`) for deep cloning objects.

Reducing Boilerplate

Redux is in part [inspired by Flux](#), and the most common complaint about Flux is how it makes you write a lot of boilerplate. In this recipe, we will consider how Redux lets us choose how verbose we'd like our code to be, depending on personal style, team preferences, longer term maintainability, and so on.

Actions

Actions are plain objects describing what happened in the app, and serve as the sole way to describe an intention to mutate the data. It's important that **actions being objects you have to dispatch is not boilerplate, but one of the [fundamental design choices](#) of Redux**.

There are frameworks claiming to be similar to Flux, but without a concept of [action](#) objects. In terms of being predictable, this is a step backwards from Flux or Redux. If there are no serializable plain object actions, it is impossible to record and replay user sessions, or to implement [hot reloading with time travel](#). If you'd rather modify data directly, you don't need Redux.

Actions look like this:

```
{ type: 'ADD_TODO', text: 'Use Redux' }
{ type: 'REMOVE_TODO', id: 42 }
{ type: 'LOAD_ARTICLE', response: { ... } }
```

It is a common convention that actions have a constant type that helps reducers (or Stores in Flux) identify them. We recommend that you use strings and not [Symbols](#) for [action](#) types, because strings are serializable, and by using Symbols you make recording and replaying harder than it needs to be.

In Flux, it is traditionally thought that you would define every [action](#) type as a string constant:

```
const ADD_TODO = 'ADD_TODO'
const REMOVE_TODO = 'REMOVE_TODO'
const LOAD_ARTICLE = 'LOAD_ARTICLE'
```

Why is this beneficial? **It is often claimed that constants are unnecessary, and for small projects, this might be correct.** For larger projects, there are some benefits to defining [action](#) types as constants:

- It helps keep the naming consistent because all [action](#) types are gathered in a single place.
- Sometimes you want to see all existing actions before working on a new feature. It may be that the [action](#) you need was already added by somebody on the team, but you didn't know.
- The list of [action](#) types that were added, removed, and changed in a Pull Request helps everyone on the team keep track of scope and implementation of new features.
- If you make a typo when importing an [action](#) constant, you will get `undefined`. Redux will immediately throw when dispatching such an [action](#), and you'll find the mistake sooner.

It is up to you to choose the conventions for your project. You may start by using inline strings, and later transition to constants, and maybe later group them into a single file. Redux does not have any opinion here, so use your best judgment.

Action Creators

It is another common convention that, instead of creating [action](#) objects inline in the places where you dispatch the actions, you would create functions generating them.

For example, instead of calling `dispatch` with an object literal:

```
// somewhere in an event handler
dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})
```

You might write an [action](#) creator in a separate file, and import it into your component:

actionCreators.js

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

AddTodo.js

```
import { addTodo } from './actionCreators'

// somewhere in an event handler
dispatch(addTodo('Use Redux'))
```

[Action](#) creators have often been criticized as boilerplate. Well, you don't have to write them! **You can use object literals if you feel this better suits your project.** There are, however, some benefits for writing [action](#) creators you should know about.

Let's say a designer comes back to us after reviewing our prototype, and tells us that we need to allow three todos maximum. We can enforce this by rewriting our [action](#) creator to a callback form with [redux-thunk middleware](#) and adding an early exit:

```
function addTodoWithoutCheck(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

export function addTodo(text) {
  // This form is allowed by Redux Thunk middleware
  // described below in "Async Action Creators" section.
  return function(dispatch, getState) {
    if (getState().todos.length === 3) {
      // Exit early
      return
    }
    dispatch(addTodoWithoutCheck(text))
  }
}
```

```
}
```

We just modified how the `addTodo` [action](#) creator behaves, completely invisible to the calling code. **We don't have to worry about looking at each place where todos are being added, to make sure they have this check.** [Action](#) creators let you decouple additional logic around dispatching an [action](#), from the actual components emitting those actions. It's very handy when the application is under heavy development, and the requirements change often.

Generating Action Creators

Some frameworks like [Flummox](#) generate [action](#) type constants automatically from the [action](#) creator function definitions. The idea is that you don't need to both define `ADD_TODO` constant and `addTodo()` [action](#) creator. Under the hood, such solutions still generate [action](#) type constants, but they're created implicitly so it's a level of indirection and can cause confusion. We recommend creating your [action](#) type constants explicitly.

Writing simple [action](#) creators can be tiresome and often ends up generating redundant boilerplate code:

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

export function editTodo(id, text) {
  return {
    type: 'EDIT_TODO',
    id,
    text
  }
}

export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  }
}
```

You can always write a function that generates an [action](#) creator:

```
function makeActionCreator(type, ...argNames) {
  return function(...args) {
    const action = { type }
    argNames.forEach((arg, index) => {
      action[argNames[index]] = args[index]
    })
    return action
  }
}

const ADD_TODO = 'ADD_TODO'
const EDIT_TODO = 'EDIT_TODO'
const REMOVE_TODO = 'REMOVE_TODO'

export const addTodo = makeActionCreator(ADD_TODO, 'text')
export const editTodo = makeActionCreator(EDIT_TODO, 'id', 'text')
export const removeTodo = makeActionCreator(REMOVE_TODO, 'id')
```

There are also utility libraries to aid in generating [action](#) creators, such as [redux-act](#) and [redux-actions](#). These can help reduce boilerplate code and enforce adherence to standards such as [Flux Standard Action \(FSA\)](#).

Async Action Creators

[Middleware](#) lets you inject custom logic that interprets every [action](#) object before it is dispatched. Async actions are the most common use case for [middleware](#).

Without any [middleware](#), `dispatch` only accepts a plain object, so we have to perform AJAX calls inside our components:

actionCreators.js

```
export function loadPostsSuccess(userId, response) {
  return {
    type: 'LOAD_POSTS_SUCCESS',
    userId,
    response
  }
}

export function loadPostsFailure(userId, error) {
  return {
    type: 'LOAD_POSTS_FAILURE',
    userId,
    error
  }
}

export function loadPostsRequest(userId) {
  return {
    type: 'LOAD_POSTS_REQUEST',
    userId
  }
}
```

UserInfo.js

```
import { Component } from 'react'
import { connect } from 'react-redux'
import {
  loadPostsRequest,
  loadPostsSuccess,
  loadPostsFailure
} from './actionCreators'

class Posts extends Component {
  loadData(userId) {
    // Injected into props by React Redux `connect()` call:
    const { dispatch, posts } = this.props

    if (posts[userId]) {
      // There is cached data! Don't do anything.
      return
    }

    // Reducer can react to this action by setting
    // `isFetching` and thus letting us show a spinner.
    dispatch(loadPostsRequest(userId))
  }
}
```

```

// Reducer can react to these actions by filling the `users`.
fetch(`http://myapi.com/users/${userId}/posts`).then(
  response => dispatch(loadPostsSuccess(userId, response)),
  error => dispatch(loadPostsFailure(userId, error))
)
}

componentDidMount() {
  this.loadData(this.props.userId)
}

componentDidUpdate(prevProps) {
  if (prevProps.userId !== this.props.userId) {
    this.loadData(this.props.userId)
  }
}

render() {
  if (this.props.isFetching) {
    return <p>Loading...</p>
  }

  const posts = this.props.posts.map(post => (
    <Post post={post} key={post.id} />
  ))

  return <div>{posts}</div>
}
}

export default connect(state => ({
  posts: state.posts,
  isFetching: state.isFetching
}))(Posts)

```

However, this quickly gets repetitive because different components request data from the same API endpoints. Moreover, we want to reuse some of this logic (e.g., early exit when there is cached data available) from many components.

Middleware lets us write more expressive, potentially async **action creators**. It lets us dispatch something other than plain objects, and interprets the values. For example, **middleware** can “catch” dispatched Promises and turn them into a pair of request and success/failure actions.

The simplest example of **middleware** is **redux-thunk**. “**Thunk**” **middleware** lets you write **action creators** as “**thunks**”, that is, functions returning functions. This inverts the control: you will get `dispatch` as an argument, so you can write an **action** creator that dispatches many times.

Note

Thunk **middleware** is just one example of **middleware**. **Middleware** is not about “letting you dispatch functions”. It’s about letting you dispatch anything that the particular **middleware** you use knows how to handle. Thunk **middleware** adds a specific behavior when you dispatch functions, but it really depends on the **middleware** you use.

Consider the code above rewritten with **redux-thunk**:

actionCreators.js

```

export function loadPosts(userId) {

```

```

// Interpreted by the thunk middleware:
return function(dispatch, getState) {
  const { posts } = getState()
  if (posts[userId]) {
    // There is cached data! Don't do anything.
    return
  }

  dispatch({
    type: 'LOAD_POSTS_REQUEST',
    userId
  })

  // Dispatch vanilla actions asynchronously
  fetch(`http://myapi.com/users/${userId}/posts`).then(
    response =>
      dispatch({
        type: 'LOAD_POSTS_SUCCESS',
        userId,
        response
      }),
    error =>
      dispatch({
        type: 'LOAD_POSTS_FAILURE',
        userId,
        error
      })
  )
}
}
}

```

UserInfo.js

```

import { Component } from 'react'
import { connect } from 'react-redux'
import { loadPosts } from './actionCreators'

class Posts extends Component {
  componentDidMount() {
    this.props.dispatch(loadPosts(this.props.userId))
  }

  componentDidUpdate(prevProps) {
    if (prevProps.userId !== this.props.userId) {
      this.props.dispatch(loadPosts(this.props.userId))
    }
  }

  render() {
    if (this.props.isFetching) {
      return <p>Loading...</p>
    }

    const posts = this.props.posts.map(post => (
      <Post post={post} key={post.id} />
    ))

    return <div>{posts}</div>
  }
}

export default connect(state => ({
  posts: state.posts,
  isFetching: state.isFetching

```

```
}})(Posts)
```

This is much less typing! If you'd like, you can still have “vanilla” [action](#) creators like `loadPostsSuccess` which you'd use from a container `loadPosts` [action](#) creator.

Finally, you can write your own [middleware](#). Let's say you want to generalize the pattern above and describe your async [action](#) creators like this instead:

```
export function loadPosts(userId) {
  return {
    // Types of actions to emit before and after
    types: ['LOAD_POSTS_REQUEST', 'LOAD_POSTS_SUCCESS', 'LOAD_POSTS_FAILURE'],
    // Check the cache (optional):
    shouldCallAPI: state => !state.posts[userId],
    // Perform the fetching:
    callAPI: () => fetch(`http://myapi.com/users/${userId}/posts`),
    // Arguments to inject in begin/end actions
    payload: { userId }
  }
}
```

The [middleware](#) that interprets such actions could look like this:

```
function callAPIMiddleware({ dispatch, getState }) {
  return next => action => {
    const { types, callAPI, shouldCallAPI = () => true, payload = {} } = action

    if (!types) {
      // Normal action: pass it on
      return next(action)
    }

    if (
      !Array.isArray(types) ||
      types.length !== 3 ||
      !types.every(type => typeof type === 'string')
    ) {
      throw new Error('Expected an array of three string types.')
    }

    if (typeof callAPI !== 'function') {
      throw new Error('Expected callAPI to be a function.')
    }

    if (!shouldCallAPI(getState())) {
      return
    }

    const [requestType, successType, failureType] = types

    dispatch(
      Object.assign({}, payload, {
        type: requestType
      })
    )

    return callAPI().then(
      response =>
        dispatch(
          Object.assign({}, payload, {
            response,
            type: successType
          })
        )
    )
  }
}
```

```

    })
  ),
  error =>
    dispatch(
      Object.assign({}, payload, {
        error,
        type: failureType
      })
    )
  )
}
}
}

```

After passing it once to `applyMiddleware(...middlewares)`, you can write all your API-calling `action` creators the same way:

```

export function loadPosts(userId) {
  return {
    types: ['LOAD_POSTS_REQUEST', 'LOAD_POSTS_SUCCESS', 'LOAD_POSTS_FAILURE'],
    shouldCallAPI: state => !state.posts[userId],
    callAPI: () => fetch(`http://myapi.com/users/${userId}/posts`),
    payload: { userId }
  }
}

export function loadComments(postId) {
  return {
    types: [
      'LOAD_COMMENTS_REQUEST',
      'LOAD_COMMENTS_SUCCESS',
      'LOAD_COMMENTS_FAILURE'
    ],
    shouldCallAPI: state => !state.comments[postId],
    callAPI: () => fetch(`http://myapi.com/posts/${postId}/comments`),
    payload: { postId }
  }
}

export function addComment(postId, message) {
  return {
    types: [
      'ADD_COMMENT_REQUEST',
      'ADD_COMMENT_SUCCESS',
      'ADD_COMMENT_FAILURE'
    ],
    callAPI: () =>
      fetch(`http://myapi.com/posts/${postId}/comments`, {
        method: 'post',
        headers: {
          Accept: 'application/json',
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ message })
      }),
    payload: { postId, message }
  }
}

```

Reducers

Redux reduces the boilerplate of Flux stores considerably by describing the update logic as a function. A function is simpler than an object, and much simpler than a class.

Consider this Flux [store](#):

```
const _todos = []

const TodoStore = Object.assign({}, EventEmitter.prototype, {
  getAll() {
    return _todos
  }
})

AppDispatcher.register(function(action) {
  switch (action.type) {
    case ActionTypes.ADD_TODO:
      const text = action.text.trim()
      _todos.push(text)
      TodoStore.emitChange()
  }
})

export default TodoStore
```

With Redux, the same update logic can be described as a reducing function:

```
export function todos(state = [], action) {
  switch (action.type) {
    case ActionTypes.ADD_TODO:
      const text = action.text.trim()
      return [...state, text]
    default:
      return state
  }
}
```

The `switch` statement is *not* the real boilerplate. The real boilerplate of Flux is conceptual: the need to emit an update, the need to register the [Store](#) with a Dispatcher, the need for the [Store](#) to be an object (and the complications that arise when you want a universal app).

It's unfortunate that many still choose Flux framework based on whether it uses `switch` statements in the documentation. If you don't like `switch`, you can solve this with a single function, as we show below.

Generating Reducers

Let's write a function that lets us express reducers as an object mapping from [action](#) types to handlers. For example, if we want our `todos` reducers to be defined like this:

```
export const todos = createReducer([], {
  [ActionTypes.ADD_TODO]: (state, action) => {
    const text = action.text.trim()
    return [...state, text]
  }
})
```

We can write the following helper to accomplish this:

```
function createReducer(initialState, handlers) {
```

```
return function reducer(state = initialState, action) {  
  if (handlers.hasOwnProperty(action.type)) {  
    return handlers[action.type](state, action)  
  } else {  
    return state  
  }  
}  
}
```

This wasn't difficult, was it? Redux doesn't provide such a helper function by default because there are many ways to write it. Maybe you want it to automatically convert plain JS objects to Immutable objects to hydrate the server [state](#). Maybe you want to merge the returned [state](#) with the current [state](#). There may be different approaches to a “catch all” handler. All of this depends on the conventions you choose for your team on a specific project.

The Redux [reducer](#) API is `(state, action) => newState`, but how you create those reducers is up to you.

Server Rendering

The most common use case for server-side rendering is to handle the *initial render* when a user (or search engine crawler) first requests our app. When the server receives the request, it renders the required component(s) into an HTML string, and then sends it as a response to the client. From that point on, the client takes over rendering duties.

We will use React in the examples below, but the same techniques can be used with other view frameworks that can render on the server.

Redux on the Server

When using Redux with server rendering, we must also send the [state](#) of our app along in our response, so the client can use it as the initial [state](#). This is important because, if we preload any data before generating the HTML, we want the client to also have access to this data. Otherwise, the markup generated on the client won't match the server markup, and the client would have to load the data again.

To send the data down to the client, we need to:

- create a fresh, new Redux [store](#) instance on every request;
- optionally dispatch some actions;
- pull the [state](#) out of [store](#);
- and then pass the [state](#) along to the client.

On the client side, a new Redux [store](#) will be created and initialized with the [state](#) provided from the server. Redux's *only* job on the server side is to provide the **initial [state](#)** of our app.

Setting Up

In the following recipe, we are going to look at how to set up server-side rendering. We'll use the simplistic [Counter app](#) as a guide and show how the server can render [state](#) ahead of time based on the request.

Install Packages

For this example, we'll be using [Express](#) as a simple web server. We also need to install the React bindings for Redux, since they are not included in Redux by default.

```
npm install --save express react-redux
```

The Server Side

The following is the outline for what our server side is going to look like. We are going to set up an [Express middleware](#) using [app.use](#) to handle all requests that come in to our server. If you're unfamiliar with Express or [middleware](#), just know that our `handleRender` function will be called every time the server receives a request.

Additionally, as we are using ES6 and JSX syntax, we will need to compile with [Babel](#) (see [this example of a Node Server with Babel](#)) and the [React preset](#).

server.js

```
import path from 'path'
import Express from 'express'
import React from 'react'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import counterApp from './reducers'
import App from './containers/App'

const app = Express()
const port = 3000

//Serve static files
app.use('/static', Express.static('static'))

// This is fired every time the server side receives a request
app.use(handleRender)

// We are going to fill these out in the sections to follow
function handleRender(req, res) {
  /* ... */
}
function renderFullPage(html, preloadedState) {
  /* ... */
}

app.listen(port)
```

Handling the Request

The first thing that we need to do on every request is to create a new Redux [store](#) instance. The only purpose of this [store](#) instance is to provide the initial [state](#) of our application.

When rendering, we will wrap `<App />`, our root component, inside a `<Provider>` to make the [store](#) available to all components in the component tree, as we saw in [Usage with React](#).

The key step in server side rendering is to render the initial HTML of our component **before** we send it to the client side. To do this, we use `ReactDOMServer.renderToString()`.

We then get the initial [state](#) from our Redux [store](#) using `store.getState()`. We will see how this is passed along in our `renderFullPage` function.

```
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // Create a new Redux store instance
  const store = createStore(counterApp)

  // Render the component to a string
  const html = renderToString(
    <Provider store={store}>
      <App />
    </Provider>
  )

  // Grab the initial state from our Redux store
  const preloadedState = store.getState()

  // Send the rendered page back to the client
  res.send(renderFullPage(html, preloadedState))
}
```

Inject Initial Component HTML and State

The final step on the server side is to inject our initial component HTML and initial `state` into a template to be rendered on the client side. To pass along the `state`, we add a `<script>` tag that will attach `preloadedState` to `window.__PRELOADED_STATE__`.

The `preloadedState` will then be available on the client side by accessing `window.__PRELOADED_STATE__`.

We also include our bundle file for the client-side application via a script tag. This is whatever output your bundling tool provides for your client entry point. It may be a static file or a URL to a hot reloading development server.

```
function renderFullPage(html, preloadedState) {
  return `
    <!doctype html>
    <html>
      <head>
        <title>Redux Universal Example</title>
      </head>
      <body>
        <div id="root">${html}</div>
        <script>
          // WARNING: See the following for security issues around embedding JSON in HTML:
          // http://redux.js.org/recipes/ServerRendering.html#security-considerations
          window.__PRELOADED_STATE__ = ${JSON.stringify(preloadedState).replace(
            /</g,
            '\\u003c'
          )}
        </script>
        <script src="/static/bundle.js"></script>
      </body>
    </html>
  `
}
```

The Client Side

The client side is very straightforward. All we need to do is grab the initial `state` from `window.__PRELOADED_STATE__`, and pass it to our `createStore()` function as the initial `state`.

Let's take a look at our new client file:

client.js

```
import React from 'react'
import { hydrate } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './containers/App'
import counterApp from './reducers'

// Grab the state from a global variable injected into the server-generated HTML
const preloadedState = window.__PRELOADED_STATE__

// Allow the passed state to be garbage-collected
delete window.__PRELOADED_STATE__
```

```
// Create Redux store with initial state
const store = createStore(counterApp, preloadedState)

hydrate(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

You can set up your build tool of choice (Webpack, Browserify, etc.) to compile a bundle file into

```
static/bundle.js .
```

When the page loads, the bundle file will be started up and `ReactDOM.hydrate()` will reuse the server-rendered HTML. This will connect our newly-started React instance to the virtual DOM used on the server. Since we have the same initial `state` for our Redux `store` and used the same code for all our view components, the result will be the same real DOM.

And that's it! That is all we need to do to implement server side rendering.

But the result is pretty vanilla. It essentially renders a static view from dynamic code. What we need to do next is build an initial `state` dynamically to allow that rendered view to be dynamic.

Preparing the Initial State

Because the client side executes ongoing code, it can start with an empty initial `state` and obtain any necessary `state` on demand and over time. On the server side, rendering is synchronous and we only get one shot to render our view. We need to be able to compile our initial `state` during the request, which will have to react to input and obtain external `state` (such as that from an API or database).

Processing Request Parameters

The only input for server side code is the request made when loading up a page in your app in your browser. You may choose to configure the server during its boot (such as when you are running in a development vs. production environment), but that configuration is static.

The request contains information about the URL requested, including any query parameters, which will be useful when using something like [React Router](#). It can also contain headers with inputs like cookies or authorization, or POST body data. Let's see how we can set the initial counter `state` based on a query parameter.

server.js

```
import qs from 'qs' // Add this at the top of the file
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // Read the counter from the request, if provided
  const params = qs.parse(req.query)
  const counter = parseInt(params.counter, 10) || 0

  // Compile an initial state
  let preloadedState = { counter }

  // Create a new Redux store instance
  const store = createStore(counterApp, preloadedState)
```

```

// Render the component to a string
const html = renderToString(
  <Provider store={store}>
    <App />
  </Provider>
)

// Grab the initial state from our Redux store
const finalState = store.getState()

// Send the rendered page back to the client
res.send(renderFullPage(html, finalState))
}

```

The code reads from the Express `Request` object passed into our server [middleware](#). The parameter is parsed into a number and then set in the initial [state](#). If you visit <http://localhost:3000/?counter=100> in your browser, you'll see the counter starts at 100. In the rendered HTML, you'll see the counter output as 100 and the `__PRELOADED_STATE__` variable has the counter set in it.

Async State Fetching

The most common issue with server side rendering is dealing with [state](#) that comes in asynchronously. Rendering on the server is synchronous by nature, so it's necessary to map any asynchronous fetches into a synchronous operation.

The easiest way to do this is to pass through some callback back to your synchronous code. In this case, that will be a function that will reference the response object and send back our rendered HTML to the client. Don't worry, it's not as hard as it may sound.

For our example, we'll imagine there is an external datastore that contains the counter's initial value (Counter As A Service, or CaaS). We'll make a mock call over to them and build our initial [state](#) from the result. We'll start by building out our API call:

api/counter.js

```

function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min
}

export function fetchCounter(callback) {
  setTimeout(() => {
    callback(getRandomInt(1, 100))
  }, 500)
}

```

Again, this is just a mock API, so we use `setTimeout` to simulate a network request that takes 500 milliseconds to respond (this should be much faster with a real world API). We pass in a callback that returns a random number asynchronously. If you're using a Promise-based API client, then you would issue this callback in your `then` handler.

On the server side, we simply wrap our existing code in the `fetchCounter` and receive the result in the callback:

server.js

```

// Add this to our imports
import { fetchCounter } from './api/counter'

```

```
import { renderToString } from 'react-dom/server'

function handleRender(req, res) {
  // Query our mock API asynchronously
  fetchCounter(apiResult => {
    // Read the counter from the request, if provided
    const params = qs.parse(req.query)
    const counter = parseInt(params.counter, 10) || apiResult || 0

    // Compile an initial state
    let preloadedState = { counter }

    // Create a new Redux store instance
    const store = createStore(counterApp, preloadedState)

    // Render the component to a string
    const html = renderToString(
      <Provider store={store}>
        <App />
      </Provider>
    )

    // Grab the initial state from our Redux store
    const finalState = store.getState()

    // Send the rendered page back to the client
    res.send(renderFullPage(html, finalState))
  })
}
```

Because we call `res.send()` inside of the callback, the server will hold open the connection and won't send any data until that callback executes. You'll notice a 500ms delay is now added to each server request as a result of our new API call. A more advanced usage would handle errors in the API gracefully, such as a bad response or timeout.

Security Considerations

Because we have introduced more code that relies on user generated content (UGC) and input, we have increased our attack surface area for our application. It is important for any application that you ensure your input is properly sanitized to prevent things like cross-site scripting (XSS) attacks or code injections.

In our example, we take a rudimentary approach to security. When we obtain the parameters from the request, we use `parseInt` on the `counter` parameter to ensure this value is a number. If we did not do this, you could easily get dangerous data into the rendered HTML by providing a script tag in the request. That might look like this: `?counter=</script><script>doSomethingBad();</script>`

For our simplistic example, coercing our input into a number is sufficiently secure. If you're handling more complex input, such as freeform text, then you should run that input through an appropriate sanitization function, such as [xss-filters](#).

Furthermore, you can add additional layers of security by sanitizing your [state](#) output. `JSON.stringify` can be subject to script injections. To counter this, you can scrub the JSON string of HTML tags and other dangerous characters. This can be done with either a simple text replacement on the string, e.g.

```
JSON.stringify(state).replace(/</g, '\\u003c') , or via more sophisticated libraries such as serialize-javascript.
```

Next Steps

You may want to read [Async Actions](#) to learn more about expressing asynchronous flow in Redux with async primitives such as Promises and `thunks`. Keep in mind that anything you learn there can also be applied to universal rendering.

If you use something like [React Router](#), you might also want to express your data fetching dependencies as static `fetchData()` methods on your route handler components. They may return [async actions](#), so that your `handleRender` function can match the route to the route handler component classes, dispatch `fetchData()` result for each of them, and render only after the Promises have resolved. This way the specific API calls required for different routes are colocated with the route handler component definitions. You can also use the same technique on the client side to prevent the router from switching the page until its data has been loaded.

Writing Tests

Because most of the Redux code you write are functions, and many of them are pure, they are easy to test without mocking.

Setting Up

We recommend [Jest](#) as the testing engine. Note that it runs in a Node environment, so you won't have access to the DOM.

```
npm install --save-dev jest
```

To use it together with [Babel](#), you will need to install `babel-jest` :

```
npm install --save-dev babel-jest
```

and configure it to use [babel-preset-env](#) features in `.babelrc` :

```
{
  "presets": ["@babel/preset-env"]
}
```

Then, add this to `scripts` in your `package.json` :

```
{
  ...
  "scripts": {
    ...
    "test": "jest",
    "test:watch": "npm test -- --watch"
  },
  ...
}
```

and run `npm test` to run it once, or `npm run test:watch` to test on every file change.

Action Creators

In Redux, [action](#) creators are functions which return plain objects. When testing [action](#) creators, we want to test whether the correct [action](#) creator was called and also whether the right [action](#) was returned.

Example

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}
```

can be tested like:

```
import * as actions from '../actions/ToDoActions'
import * as types from '../constants/ActionTypes'

describe('actions', () => {
  it('should create an action to add a todo', () => {
    const text = 'Finish docs'
    const expectedAction = {
      type: types.ADD_TODO,
      text
    }
    expect(actions.addTodo(text)).toEqual(expectedAction)
  })
})
```

Async Action Creators

For async [action](#) creators using [Redux Thunk](#) or other [middleware](#), it's best to completely mock the Redux [store](#) for tests. You can apply the [middleware](#) to a mock [store](#) using [redux-mock-store](#). You can also use [fetch-mock](#) to mock the HTTP requests.

Example

```
import 'cross-fetch/polyfill'

function fetchTodosRequest() {
  return {
    type: FETCH_TODOS_REQUEST
  }
}

function fetchTodosSuccess(body) {
  return {
    type: FETCH_TODOS_SUCCESS,
    body
  }
}

function fetchTodosFailure(ex) {
  return {
    type: FETCH_TODOS_FAILURE,
    ex
  }
}

export function fetchTodos() {
  return dispatch => {
    dispatch(fetchTodosRequest())
    return fetch('http://example.com/todos')
      .then(res => res.json())
      .then(body => dispatch(fetchTodosSuccess(body)))
      .catch(ex => dispatch(fetchTodosFailure(ex)))
  }
}
```

can be tested like:

```
import configureMockStore from 'redux-mock-store'
```

```

import thunk from 'redux-thunk'
import * as actions from '../actions/ToDoActions'
import * as types from '../constants/ActionTypes'
import fetchMock from 'fetch-mock'
import expect from 'expect' // You can use any testing library

const middlewares = [thunk]
const mockStore = configureMockStore(middlewares)

describe('async actions', () => {
  afterEach(() => {
    fetchMock.restore()
  })

  it('creates FETCH_TODOS_SUCCESS when fetching todos has been done', () => {
    fetchMock.getOnce('/todos', {
      body: { todos: ['do something'] },
      headers: { 'content-type': 'application/json' }
    })

    const expectedActions = [
      { type: types.FETCH_TODOS_REQUEST },
      { type: types.FETCH_TODOS_SUCCESS, body: { todos: ['do something'] } }
    ]
    const store = mockStore({ todos: [] })

    return store.dispatch(actions.fetchTodos()).then(() => {
      // return of async actions
      expect(store.getActions()).toEqual(expectedActions)
    })
  })
})

```

Reducers

A **reducer** should return the new **state** after applying the **action** to the previous **state**, and that's the behavior tested below.

Example

```

import { ADD_TODO } from '../constants/ActionTypes'

const initialState = [
  {
    text: 'Use Redux',
    completed: false,
    id: 0
  }
]

export default function todos(state = initialState, action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        {
          id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,
          completed: false,
          text: action.text
        },
        ...state
      ]
  }
}

```

```

    default:
      return state
    }
  }
}

```

can be tested like:

```

import reducer from '../structuring-reducers/todos'
import * as types from '../constants/ActionTypes'

describe('todos reducer', () => {
  it('should return the initial state', () => {
    expect(reducer(undefined, {})).toEqual([
      {
        text: 'Use Redux',
        completed: false,
        id: 0
      }
    ])
  })

  it('should handle ADD_TODO', () => {
    expect(
      reducer([], {
        type: types.ADD_TODO,
        text: 'Run the tests'
      })
    ).toEqual([
      {
        text: 'Run the tests',
        completed: false,
        id: 0
      }
    ])

    expect(
      reducer(
        [
          {
            text: 'Use Redux',
            completed: false,
            id: 0
          }
        ],
        {
          type: types.ADD_TODO,
          text: 'Run the tests'
        }
      )
    ).toEqual([
      {
        text: 'Run the tests',
        completed: false,
        id: 1
      },
      {
        text: 'Use Redux',
        completed: false,
        id: 0
      }
    ])
  })
})

```

Components

A nice thing about React components is that they are usually small and only rely on their props. That makes them easy to test.

First, we will install [Enzyme](#). Enzyme uses the [React Test Utilities](#) underneath, but is more convenient, readable, and powerful.

```
npm install --save-dev enzyme
```

We will also need to install Enzyme adapter for our version of React. Enzyme has adapters that provide compatibility with `React 16.x`, `React 15.x`, `React 0.14.x` and `React 0.13.x`. If you are using React 16 you can run:

```
npm install --save-dev enzyme-adapter-react-16
```

To test the components we make a `setup()` helper that passes the stubbed callbacks as props and renders the component with [shallow rendering](#). This lets individual tests assert on whether the callbacks were called when expected.

Example

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import TodoTextInput from './TodoTextInput'

class Header extends Component {
  handleSave(text) {
    if (text.length !== 0) {
      this.props.addToDo(text)
    }
  }

  render() {
    return (
      <header className="header">
        <h1>todos</h1>
        <TodoTextInput
          newTodo={true}
          onSave={this.handleSave.bind(this)}
          placeholder="What needs to be done?"
        />
      </header>
    )
  }
}

Header.propTypes = {
  addToDo: PropTypes.func.isRequired
}

export default Header
```

can be tested like:

```
import React from 'react'
import Enzyme, { shallow } from 'enzyme'
import Adapter from 'enzyme-adapter-react-16'
```

```

import Header from '../components/Header'

Enzyme.configure({ adapter: new Adapter() })

function setup() {
  const props = {
    addToDo: jest.fn()
  }

  const enzymeWrapper = shallow(<Header {...props} />)

  return {
    props,
    enzymeWrapper
  }
}

describe('components', () => {
  describe('Header', () => {
    it('should render self and subcomponents', () => {
      const { enzymeWrapper } = setup()

      expect(enzymeWrapper.find('header').hasClass('header')).toBe(true)

      expect(enzymeWrapper.find('h1').text()).toBe('todos')

      const todoInputProps = enzymeWrapper.find('TodoTextInput').props()
      expect(todoInputProps.newTodo).toBe(true)
      expect(todoInputProps.placeholder).toEqual('What needs to be done?')
    })

    it('should call addToDo if length of text is greater than 0', () => {
      const { enzymeWrapper, props } = setup()
      const input = enzymeWrapper.find('TodoTextInput')
      input.props().onSave('')
      expect(props.addToDo.mock.calls.length).toBe(0)
      input.props().onSave('Use Redux')
      expect(props.addToDo.mock.calls.length).toBe(1)
    })
  })
})

```

Connected Components

If you use a library like [React Redux](#), you might be using [higher-order components](#) like `connect()`. This lets you inject Redux [state](#) into a regular React component.

Consider the following `App` component:

```

import { connect } from 'react-redux'

class App extends Component {
  /* ... */
}

export default connect(mapStateToProps)(App)

```

In a unit test, you would normally import the `App` component like this:

```

import App from './App'

```

However, when you import it, you're actually holding the wrapper component returned by `connect()`, and not the `App` component itself. If you want to test its interaction with Redux, this is good news: you can wrap it in a `<Provider>` with a `store` created specifically for this unit test. But sometimes you want to test just the rendering of the component, without a Redux `store`.

In order to be able to test the `App` component itself without having to deal with the decorator, we recommend you to also export the undecorated component:

```
import { connect } from 'react-redux'

// Use named export for unconnected component (for tests)
export class App extends Component {
  /* ... */
}

// Use default export for the connected component (for app)
export default connect(mapStateToProps)(App)
```

Since the default export is still the decorated component, the import statement pictured above will work as before so you won't have to change your application code. However, you can now import the undecorated `App` components in your test file like this:

```
// Note the curly braces: grab the named export instead of default export
import { App } from './App'
```

And if you need both:

```
import ConnectedApp, { App } from './App'
```

In the app itself, you would still import it normally:

```
import App from './App'
```

You would only use the named export for tests.

A Note on Mixing ES6 Modules and CommonJS

If you are using ES6 in your application source, but write your tests in ES5, you should know that Babel handles the interchangeable use of ES6 `import` and CommonJS `require` through its `interop` capability to run two module formats side-by-side, but the behavior is `slightly different`. If you add a second export beside your default export, you can no longer import the default using `require('./App')`. Instead you have to use `require('./App').default`.

Middleware

`Middleware` functions wrap behavior of `dispatch` calls in Redux, so to test this modified behavior we need to mock the behavior of the `dispatch` call.

Example

First, we'll need a `middleware` function. This is similar to the real `redux-thunk`.


```
const thunk = ({ dispatch, getState }) => next => action => {
  if (typeof action === 'function') {
    return action(dispatch, getState)
  }

  return next(action)
}
```

We need to create a fake `getState`, `dispatch`, and `next` functions. We use `jest.fn()` to create stubs, but with other test frameworks you would likely use [Sinon](#).

The invoke function runs our [middleware](#) in the same way Redux does.

```
const create = () => {
  const store = {
    getState: jest.fn(() => ({})),
    dispatch: jest.fn()
  }
  const next = jest.fn()

  const invoke = action => thunk(store)(next)(action)

  return { store, next, invoke }
}
```

We test that our [middleware](#) is calling the `getState`, `dispatch`, and `next` functions at the right time.

```
it('passes through non-function action', () => {
  const { next, invoke } = create()
  const action = { type: 'TEST' }
  invoke(action)
  expect(next).toHaveBeenCalledWith(action)
})

it('calls the function', () => {
  const { invoke } = create()
  const fn = jest.fn()
  invoke(fn)
  expect(fn).toHaveBeenCalled()
})

it('passes dispatch and getState', () => {
  const { store, invoke } = create()
  invoke((dispatch, getState) => {
    dispatch('TEST DISPATCH')
    getState()
  })
  expect(store.dispatch).toHaveBeenCalledWith('TEST DISPATCH')
  expect(store.getState).toHaveBeenCalled()
})
```

In some cases, you will need to modify the `create` function to use different mock implementations of `getState` and `next`.

Glossary

- [Enzyme](#): Enzyme is a JavaScript Testing utility for React that makes it easier to assert, manipulate, and traverse your React Components' output.

- [React Test Utils](#): Test Utilities for React. Used by Enzyme.
- [Shallow rendering](#): Shallow rendering lets you instantiate a component and effectively get the result of its `render` method just a single level deep instead of rendering components recursively to a DOM. Shallow rendering is useful for unit tests, where you test a particular component only, and importantly not its children. This also means that changing a child component won't affect the tests for the parent component. Testing a component and all its children can be accomplished with [Enzyme's](#) `mount()` method, aka full DOM rendering.

Computing Derived Data

[Reselect](#) is a simple library for creating memoized, composable **selector** functions. Reselect selectors can be used to efficiently compute derived data from the Redux [store](#).

Motivation for Memoized Selectors

Let's revisit the [Todos List example](#):

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case 'SHOW_ALL':
      return todos
    case 'SHOW_COMPLETED':
      return todos.filter(t => t.completed)
    case 'SHOW_ACTIVE':
      return todos.filter(t => !t.completed)
  }
}

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

In the above example, `mapStateToProps` calls `getVisibleTodos` to calculate `todos`. This works great, but there is a drawback: `todos` is calculated every time the component is updated. If the [state](#) tree is large, or the calculation expensive, repeating the calculation on every update may cause performance problems. Reselect can help to avoid these unnecessary recalculations.

Creating a Memoized Selector

We would like to replace `getVisibleTodos` with a memoized selector that recalculates `todos` when the value of `state.todos` OR `state.visibilityFilter` changes, but not when changes occur in other (unrelated) parts of the `state` tree.

Reselect provides a function `createSelector` for creating memoized selectors. `createSelector` takes an array of input-selectors and a transform function as its arguments. If the Redux `state` tree is changed in a way that causes the value of an input-selector to change, the selector will call its transform function with the values of the input-selectors as arguments and return the result. If the values of the input-selectors are the same as the previous call to the selector, it will return the previously computed value instead of calling the transform function.

Let's define a memoized selector named `getVisibleTodos` to replace the non-memoized version above:

selectors/index.js

```
import { createSelector } from 'reselect'

const getVisibilityFilter = state => state.visibilityFilter
const getTodos = state => state.todos

export const getVisibleTodos = createSelector(
  [getVisibilityFilter, getTodos],
  (visibilityFilter, todos) => {
    switch (visibilityFilter) {
      case 'SHOW_ALL':
        return todos
      case 'SHOW_COMPLETED':
        return todos.filter(t => t.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(t => !t.completed)
    }
  }
)
```

In the example above, `getVisibilityFilter` and `getTodos` are input-selectors. They are created as ordinary non-memoized selector functions because they do not transform the data they select. `getVisibleTodos` on the other hand is a memoized selector. It takes `getVisibilityFilter` and `getTodos` as input-selectors, and a transform function that calculates the filtered todos list.

Composing Selectors

A memoized selector can itself be an input-selector to another memoized selector. Here is `getVisibleTodos` being used as an input-selector to a selector that further filters the todos by keyword:

```
const getKeyword = state => state.keyword

const getVisibleTodosFilteredByKeyword = createSelector(
  [getVisibleTodos, getKeyword],
  (visibleTodos, keyword) =>
    visibleTodos.filter(todo => todo.text.indexOf(keyword) > -1)
)
```

Connecting a Selector to the Redux Store

If you are using `React Redux`, you can call selectors as regular functions inside `mapStateToProps`:

containers/VisibleTodoList.js

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { getVisibleTodos } from '../selectors'

const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Accessing React Props in Selectors

So far we have only seen selectors receive the Redux [store state](#) as an argument, but a selector can receive props too.

For this example, we're going to extend our app to handle multiple Todo lists. Our [state](#) needs to be refactored so that it holds multiple todo lists, which each have their own `todos` and `visibilityFilter` [state](#).

We also need to refactor our reducers. Now that `todos` and `visibilityFilter` live within every list's [state](#), we only need one `todoLists` [reducer](#) to manage our [state](#).

reducers/index.js

```
import { combineReducers } from 'redux'
import todoLists from './todoLists'

export default combineReducers({
  todoLists
})
```

reducers/todoLists.js

```
// Note that we're hard coding three lists here just as an example.
// In the real world, we'd have a feature to add/remove lists,
// and this would be empty initially.
const initialState = {
  1: {
    todos: [],
    visibilityFilter: 'SHOW_ALL'
  },
}
```

```

2: {
  todos: [],
  visibilityFilter: 'SHOW_ALL'
},
3: {
  todos: [],
  visibilityFilter: 'SHOW_ALL'
}
}

const addTodo = (state, action) => {
  const todoList = state[action.listId]
  const { todos } = todoList

  return {
    ...state,
    [action.listId]: {
      ...todoList,
      todos: [
        ...todos,
        {
          id: action.id,
          text: action.text,
          completed: false
        }
      ]
    }
  }
}

const toggleTodo = (state, action) => {
  const todoList = state[action.listId]
  const { todos } = todoList

  return {
    ...state,
    [action.listId]: {
      ...todoList,
      todos: todos.map(todo =>
        (todo.id === action.id)
          ? {...todo, completed: !todo.completed}
          : todo
      )
    }
  }
}

const setVisibilityFilter = (state, action) => {
  const todoList = state[action.listId]
  return {
    ...state,
    [action.listId]: {
      ...todoList,
      visibilityFilter: action.filter
    }
  }
}

export default const todoLists = (state = initialState, action) => {
  // make sure a list with the given id exists
  if (!state[action.listId]) {
    return state;
  }

  switch (action.type) {
    case 'ADD_TODO':

```

```

    return addTodo(state, action)

    case 'TOGGLE_TODO':
    return toggleTodo(state, action)

    case 'SET_VISIBILITY_FILTER':
    return setVisibilityFilter(state, action)

    default:
    return state
  }
}

```

The `todoLists` `reducer` now handles all three actions. The `action` creators will now need to be passed a `listId` :

actions/index.js

```

let nextTodoId = 0
export const addTodo = (text, listId) => ({
  type: 'ADD_TODO',
  id: nextTodoId++,
  text,
  listId
})
export const setVisibilityFilter = (filter, listId) => ({
  type: 'SET_VISIBILITY_FILTER',
  filter,
  listId
})
export const toggleTodo = (id, listId) => ({
  type: 'TOGGLE_TODO',
  id,
  listId
})
export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

```

components/ToDoList.js

```

import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'
const ToDoList = ({ todos, toggleTodo, listId }) => (
  <ul>
    {todos.map(todo => (
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => toggleTodo(todo.id, listId)}
      />
    ))}
  </ul>
)

export default ToDoList

```

Here is an `App` component that renders three `VisibleToDoList` components, each of which has a `listId` prop:

components/App.js

```
import React from 'react'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <VisibleTodoList listId="1" />
    <VisibleTodoList listId="2" />
    <VisibleTodoList listId="3" />
  </div>
)
```

Each `VisibleTodoList` container should select a different slice of the `state` depending on the value of the `listId` prop, so we'll modify `getVisibilityFilter` and `getTodos` to accept a props argument.

selectors/todoSelectors.js

```
import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>
  state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) => state.todoLists[props.listId].todos

const getVisibleTodos = createSelector(
  [getVisibilityFilter, getTodos],
  (visibilityFilter, todos) => {
    switch (visibilityFilter) {
      case 'SHOW_COMPLETED':
        return todos.filter(todo => todo.completed)
      case 'SHOW_ACTIVE':
        return todos.filter(todo => !todo.completed)
      default:
        return todos
    }
  }
)

export default getVisibleTodos
```

`props` can be passed to `getVisibleTodos` from `mapStateToProps` :

```
const mapStateToProps = (state, props) => {
  return {
    todos: getVisibleTodos(state, props)
  }
}
```

So now `getVisibleTodos` has access to `props`, and everything seems to be working fine.

But there is a problem!

Using the `getVisibleTodos` selector with multiple instances of the `visibleTodoList` container will not correctly memoize:

containers/VisibleTodoList.js


```

import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { getVisibleTodos } from '../selectors'

const mapStateToProps = (state, props) => {
  return {
    // WARNING: THE FOLLOWING SELECTOR DOES NOT CORRECTLY MEMOIZE
    todos: getVisibleTodos(state, props)
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList

```

A selector created with `createSelector` only returns the cached value when its set of arguments is the same as its previous set of arguments. If we alternate between rendering `<VisibleTodoList listId="1" />` and `<VisibleTodoList listId="2" />`, the shared selector will alternate between receiving `{listId: 1}` and `{listId: 2}` as its `props` argument. This will cause the arguments to be different on each call, so the selector will always recompute instead of returning the cached value. We'll see how to overcome this limitation in the next section.

Sharing Selectors Across Multiple Components

The examples in this section require React Redux v4.3.0 or greater

In order to share a selector across multiple `VisibleTodoList` components **and** retain memoization, each instance of the component needs its own private copy of the selector.

Let's create a function named `makeGetVisibleTodos` that returns a new copy of the `getVisibleTodos` selector each time it is called:

selectors/todoSelectors.js

```

import { createSelector } from 'reselect'

const getVisibilityFilter = (state, props) =>
  state.todoLists[props.listId].visibilityFilter

const getTodos = (state, props) => state.todoLists[props.listId].todos

const makeGetVisibleTodos = () => {
  return createSelector(
    [getVisibilityFilter, getTodos],
    (visibilityFilter, todos) => {
      switch (visibilityFilter) {
        case 'SHOW_COMPLETED':
          return todos.filter(todo => todo.completed)
      }
    }
  )
}

```

```

    case 'SHOW_ACTIVE':
      return todos.filter(todo => !todo.completed)
    default:
      return todos
  }
}
)
}

export default makeGetVisibleTodos

```

We also need a way to give each instance of a container access to its own private selector. The `mapStateToProps` argument of `connect` can help with this.

If the `mapStateToProps` argument supplied to `connect` returns a function instead of an object, it will be used to create an individual `mapStateToProps` function for each instance of the container.

In the example below `makeMapStateToProps` creates a new `getVisibleTodos` selector, and returns a `mapStateToProps` function that has exclusive access to the new selector:

```

const makeMapStateToProps = () => {
  const getVisibleTodos = makeGetVisibleTodos()
  const mapStateToProps = (state, props) => {
    return {
      todos: getVisibleTodos(state, props)
    }
  }
  return mapStateToProps
}

```

If we pass `makeMapStateToProps` to `connect`, each instance of the `VisibleTodosList` container will get its own `mapStateToProps` function with a private `getVisibleTodos` selector. Memoization will now work correctly regardless of the render order of the `VisibleTodoList` containers.

containers/VisibleTodoList.js

```

import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { makeGetVisibleTodos } from '../selectors'

const makeMapStateToProps = () => {
  const getVisibleTodos = makeGetVisibleTodos()
  const mapStateToProps = (state, props) => {
    return {
      todos: getVisibleTodos(state, props)
    }
  }
  return mapStateToProps
}

const mapDispatchToProps = dispatch => {
  return {
    onTodoClick: id => {
      dispatch(toggleTodo(id))
    }
  }
}

const VisibleTodoList = connect(

```

```
    makeMapStateToProps,  
    mapDispatchToProps  
  )(TodoList)  
  
export default VisibleTodoList
```

Next Steps

Check out the [official documentation](#) of Reselect as well as its [FAQ](#). Most Redux projects start using Reselect when they have performance problems because of too many derived computations and wasted re-renders, so make sure you are familiar with it before you build something big. It can also be useful to study [its source code](#) so you don't think it's magic.

Implementing Undo History

Building an Undo and Redo functionality into an app has traditionally required conscious effort from the developer. It is not an easy problem with classical MVC frameworks because you need to keep track of every past [state](#) by cloning all relevant models. In addition, you need to be mindful of the undo stack because the user-initiated changes should be undoable.

This means that implementing Undo and Redo in an MVC application usually forces you to rewrite parts of your application to use a specific data mutation pattern like [Command](#).

With Redux, however, implementing undo history is a breeze. There are three reasons for this:

- There are no multiple models—just a [state](#) subtree that you want to keep track of.
- The [state](#) is already immutable, and mutations are already described as discrete actions, which is close to the undo stack mental model.
- The [reducer](#) `(state, action) => state` signature makes it natural to implement generic “[reducer](#) enhancers” or “higher order reducers”. They are functions that take your [reducer](#) and enhance it with some additional functionality while preserving its signature. Undo history is exactly such a case.

Before proceeding, make sure you have worked through the [basics tutorial](#) and understand [reducer composition](#) well. This recipe will build on top of the example described in the [basics tutorial](#).

In the first part of this recipe, we will explain the underlying concepts that make Undo and Redo possible to implement in a generic way.

In the second part of this recipe, we will show how to use [Redux Undo](#) package that provides this functionality out of the box.

Show: All, [Completed](#), [Active](#).

Understanding Undo History

Designing the State Shape

Undo history is also part of your app's [state](#), and there is no reason why we should approach it differently. Regardless of the type of the [state](#) changing over time, when you implement Undo and Redo, you want to keep track of the *history* of this [state](#) at different points in time.

For example, the [state](#) shape of a counter app might look like this:

```
{
  counter: 10
}
```

If we wanted to implement Undo and Redo in such an app, we'd need to [store](#) more [state](#) so we can answer the following questions:

- Is there anything left to undo or redo?
- What is the current [state](#)?
- What are the past (and future) states in the undo stack?

It is reasonable to suggest that our [state](#) shape should change to answer these questions:

```
{
  counter: {
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    present: 10,
    future: []
  }
}
```

Now, if user presses “Undo”, we want it to change to move into the past:

```
{
  counter: {
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8],
    present: 9,
    future: [10]
  }
}
```

And further yet:

```
{
  counter: {
    past: [0, 1, 2, 3, 4, 5, 6, 7],
    present: 8,
    future: [9, 10]
  }
}
```

When the user presses “Redo”, we want to move one step back into the future:

```
{
  counter: {
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8],
    present: 9,
    future: [10]
  }
}
```

Finally, if the user performs an [action](#) (e.g. decrement the counter) while we're in the middle of the undo stack, we're going to discard the existing future:

```
{
  counter: {
    past: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    present: 8,
    future: []
  }
}
```

The interesting part here is that it does not matter whether we want to keep an undo stack of numbers, strings, arrays, or objects. The structure will always be the same:

```
{
  counter: {
    past: [0, 1, 2],
    present: 3,
    future: [4]
  }
}
```

```
{
  todos: {
    past: [
      [],
      [{ text: 'Use Redux' }],
      [{ text: 'Use Redux', complete: true }]
    ],
    present: [
      { text: 'Use Redux', complete: true },
      { text: 'Implement Undo' }
    ],
    future: [
      [
        { text: 'Use Redux', complete: true },
        { text: 'Implement Undo', complete: true }
      ]
    ]
  }
}
```

In general, it looks like this:

```
{
  past: Array<T>,
  present: T,
  future: Array<T>
}
```

It is also up to us whether to keep a single top-level history:

```
{
  past: [
    { counterA: 1, counterB: 1 },
    { counterA: 1, counterB: 0 },
    { counterA: 0, counterB: 0 }
  ],
  present: { counterA: 2, counterB: 1 },
  future: []
}
```

Or many granular histories so user can undo and redo actions in them independently:

```
{
  counterA: {
    past: [1, 0],
    present: 2,
  }
}
```

```

    future: []
  },
  counterB: {
    past: [0],
    present: 1,
    future: []
  }
}

```

We will see later how the approach we take lets us choose how granular Undo and Redo need to be.

Designing the Algorithm

Regardless of the specific data type, the shape of the undo history `state` is the same:

```

{
  past: Array<T>,
  present: T,
  future: Array<T>
}

```

Let's talk through the algorithm to manipulate the `state` shape described above. We can define two actions to operate on this `state`: `UNDO` and `REDO`. In our `reducer`, we will do the following steps to handle these actions:

Handling Undo

- Remove the *last* element from the `past`.
- Set the `present` to the element we removed in the previous step.
- Insert the old `present` `state` at the *beginning* of the `future`.

Handling Redo

- Remove the *first* element from the `future`.
- Set the `present` to the element we removed in the previous step.
- Insert the old `present` `state` at the *end* of the `past`.

Handling Other Actions

- Insert the `present` at the end of the `past`.
- Set the `present` to the new `state` after handling the `action`.
- Clear the `future`.

First Attempt: Writing a Reducer

```

const initialState = {
  past: [],
  present: null, // (?) How do we initialize the present?
  future: []
}

function undoable(state = initialState, action) {
  const { past, present, future } = state

  switch (action.type) {

```

```

case 'UNDO':
  const previous = past[past.length - 1]
  const newPast = past.slice(0, past.length - 1)
  return {
    past: newPast,
    present: previous,
    future: [present, ...future]
  }
case 'REDO':
  const next = future[0]
  const newFuture = future.slice(1)
  return {
    past: [...past, present],
    present: next,
    future: newFuture
  }
default:
  // (?) How do we handle other actions?
  return state
}
}

```

This implementation isn't usable because it leaves out three important questions:

- Where do we get the initial `present state` from? We don't seem to know it beforehand.
- Where do we react to the external actions to save the `present` to the `past`?
- How do we actually delegate the control over the `present state` to a custom `reducer`?

It seems that `reducer` isn't the right abstraction, but we're very close.

Meet Reducer Enhancers

You might be familiar with [higher order functions](#). If you use React, you might be familiar with [higher order components](#). Here is a variation on the same pattern, applied to reducers.

A `reducer enhancer` (or a *higher order reducer*) is a function that takes a `reducer`, and returns a new `reducer` that is able to handle new actions, or to hold more `state`, delegating control to the inner `reducer` for the actions it doesn't understand. This isn't a new pattern—technically, `combineReducers()` is also a `reducer enhancer` because it takes reducers and returns a new `reducer`.

A `reducer enhancer` that doesn't do anything looks like this:

```

function doNothingWith(reducer) {
  return function(state, action) {
    // Just call the passed reducer
    return reducer(state, action)
  }
}

```

A `reducer enhancer` that combines other reducers might look like this:

```

function combineReducers(reducers) {
  return function(state = {}, action) {
    return Object.keys(reducers).reduce((nextState, key) => {
      // Call every reducer with the part of the state it manages
      nextState[key] = reducers[key](state[key], action)
      return nextState
    }, {})
  }
}

```



```
}
```

Second Attempt: Writing a Reducer Enhancer

Now that we have a better understanding of `reducer` enhancers, we can see that this is exactly what `undoable` should have been:

```
function undoable(reducer) {
  // Call the reducer with empty action to populate the initial state
  const initialState = {
    past: [],
    present: reducer(undefined, {}),
    future: []
  }

  // Return a reducer that handles undo and redo
  return function(state = initialState, action) {
    const { past, present, future } = state

    switch (action.type) {
      case 'UNDO':
        const previous = past[past.length - 1]
        const newPast = past.slice(0, past.length - 1)
        return {
          past: newPast,
          present: previous,
          future: [present, ...future]
        }
      case 'REDO':
        const next = future[0]
        const newFuture = future.slice(1)
        return {
          past: [...past, present],
          present: next,
          future: newFuture
        }
      default:
        // Delegate handling the action to the passed reducer
        const newPresent = reducer(present, action)
        if (present === newPresent) {
          return state
        }
        return {
          past: [...past, present],
          present: newPresent,
          future: []
        }
    }
  }
}
```

We can now wrap any `reducer` into `undoable` `reducer` enhancer to teach it to react to `UNDO` and `REDO` actions.

```
// This is a reducer
function todos(state = [], action) {
  /* ... */
}

// This is also a reducer!
const undoableTodos = undoable(todos)

import { createStore } from 'redux'
```

```
const store = createStore(undoableTodos)

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})

store.dispatch({
  type: 'ADD_TODO',
  text: 'Implement Undo'
})

store.dispatch({
  type: 'UNDO'
})
```

There is an important gotcha: you need to remember to append `.present` to the current `state` when you retrieve it. You may also check `.past.length` and `.future.length` to determine whether to enable or to disable the Undo and Redo buttons, respectively.

You might have heard that Redux was influenced by [Elm Architecture](#). It shouldn't come as a surprise that this example is very similar to [elm-undo-redo package](#).

Using Redux Undo

This was all very informative, but can't we just drop a library and use it instead of implementing `undoable` ourselves? Sure, we can! Meet [Redux Undo](#), a library that provides simple Undo and Redo functionality for any part of your Redux tree.

In this part of the recipe, you will learn how to make the [Todo List example](#) undoable. You can find the full source of this recipe in the [todos-with-undo example that comes with Redux](#).

Installation

First of all, you need to run

```
npm install --save redux-undo
```

This installs the package that provides the `undoable` [reducer](#) enhancer.

Wrapping the Reducer

You will need to wrap the [reducer](#) you wish to enhance with `undoable` function. For example, if you exported a `todos` [reducer](#) from a dedicated file, you will want to change it to export the result of calling `undoable()` with the [reducer](#) you wrote:

reducers/todos.js

```
import undoable, { distinctState } from 'redux-undo'

/* ... */

const todos = (state = [], action) => {
  /* ... */
```

```

}

const undoableTodos = undoable(todos, {
  filter: distinctState()
})

export default undoableTodos

```

The `distinctState()` filter serves to ignore the actions that didn't result in a [state](#) change. There are [many other options](#) to configure your undoable [reducer](#), like setting the [action](#) type for Undo and Redo actions.

Note that your `combineReducers()` call will stay exactly as it was, but the `todos` [reducer](#) will now refer to the [reducer](#) enhanced with Redux Undo:

reducers/index.js

```

import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})

export default todoApp

```

You may wrap one or more reducers in `undoable` at any level of the [reducer](#) composition hierarchy. We choose to wrap `todos` instead of the top-level combined [reducer](#) so that changes to `visibilityFilter` are not reflected in the undo history.

Updating the Selectors

Now the `todos` part of the [state](#) looks like this:

```

{
  visibilityFilter: 'SHOW_ALL',
  todos: {
    past: [
      [],
      [{ text: 'Use Redux' }],
      [{ text: 'Use Redux', complete: true }]
    ],
    present: [
      { text: 'Use Redux', complete: true },
      { text: 'Implement Undo' }
    ],
    future: [
      [
        { text: 'Use Redux', complete: true },
        { text: 'Implement Undo', complete: true }
      ]
    ]
  }
}

```

This means you need to access your [state](#) with `state.todos.present` instead of just `state.todos` :

containers/VisibleTodoList.js

```
const mapStateToProps = state => {
  return {
    todos: getVisibleTodos(state.todos.present, state.visibilityFilter)
  }
}
```

Adding the Buttons

Now all you need to do is add the buttons for the Undo and Redo actions.

First, create a new container component called `UndoRedo` for these buttons. We won't bother to split the presentational part into a separate file because it is very small:

containers/UndoRedo.js

```
import React from 'react'

/* ... */

let UndoRedo = ({ canUndo, canRedo, onUndo, onRedo }) => (
  <p>
    <button onClick={onUndo} disabled={!canUndo}>
      Undo
    </button>
    <button onClick={onRedo} disabled={!canRedo}>
      Redo
    </button>
  </p>
)
```

You will use `connect()` from [React Redux](#) to generate a container component. To determine whether to enable Undo and Redo buttons, you can check `state.todos.past.length` and `state.todos.future.length`. You won't need to write [action](#) creators for performing undo and redo because Redux Undo already provides them:

containers/UndoRedo.js

```
/* ... */

import { ActionCreators as UndoActionCreators } from 'redux-undo'
import { connect } from 'react-redux'

/* ... */

const mapStateToProps = state => {
  return {
    canUndo: state.todos.past.length > 0,
    canRedo: state.todos.future.length > 0
  }
}

const mapDispatchToProps = dispatch => {
  return {
    onUndo: () => dispatch(UndoActionCreators.undo()),
    onRedo: () => dispatch(UndoActionCreators.redo())
  }
}
```

```
UndoRedo = connect(
  mapStateToProps,
  mapDispatchToProps
)(UndoRedo)

export default UndoRedo
```

Now you can add `UndoRedo` component to the `App` component:

components/App.js

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'
import UndoRedo from '../containers/UndoRedo'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
    <UndoRedo />
  </div>
)

export default App
```

This is it! Run `npm install` and `npm start` in the [example folder](#) and try it out!

Isolating Redux Sub-Apps

Consider the case of a “big” app (contained in a `<BigApp>` component) that embeds smaller “sub-apps” (contained in `<SubApp>` components):

```
import React, { Component } from 'react'
import SubApp from './subapp'

class BigApp extends Component {
  render() {
    return (
      <div>
        <SubApp />
        <SubApp />
        <SubApp />
      </div>
    )
  }
}
```

These `<SubApp>` s will be completely independent. They won't share data or actions, and won't see or communicate with each other.

It's best not to mix this approach with standard Redux [reducer](#) composition. For typical web apps, stick with [reducer](#) composition. For “product hubs”, “dashboards”, or enterprise software that groups disparate tools into a unified package, give the sub-app approach a try.

The sub-app approach is also useful for large teams that are divided by product or feature verticals. These teams can ship sub-apps independently or in combination with an enclosing “app shell”.

Below is a sub-app's root connected component. As usual, it can render more components, connected or not, as children. Usually we'd render it in `<Provider>` and be done with it.

```
class App extends Component { ... }
export default connect(mapStateToProps)(App)
```

However, we don't have to call `ReactDOM.render(<Provider><App /></Provider>)` if we're interested in hiding the fact that the sub-app component is a Redux app.

Maybe we want to be able to run multiple instances of it in the same “bigger” app and keep it as a complete black box, with Redux being an implementation detail.

To hide Redux behind a React API, we can wrap it in a special component that initializes the [store](#) in the constructor:

```
import React, { Component } from 'react'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import reducer from './reducers'
import App from './App'

class SubApp extends Component {
  constructor(props) {
    super(props)
    this.store = createStore(reducer)
  }
}
```

```
}  
  
render() {  
  return (  
    <Provider store={this.store}>  
      <App />  
    </Provider>  
  )  
}  
}
```

This way every instance will be independent.

This pattern is *not* recommended for parts of the same app that share data. However, it can be useful when the bigger app has zero access to the smaller apps' internals, and we'd like to keep the fact that they are implemented with Redux as an implementation detail. Each component instance will have its own [store](#), so they won't "know" about each other.

Structuring Reducers

At its core, Redux is really a fairly simple design pattern: all your "write" logic goes into a single function, and the only way to run that logic is to give Redux a plain object that describes something that has happened. The Redux [store](#) calls that write logic function and passes in the current [state](#) tree and the descriptive object, the write logic function returns some new [state](#) tree, and the Redux [store](#) notifies any subscribers that the [state](#) tree has changed.

Redux puts some basic constraints on how that write logic function should work. As described in [Reducers](#), it has to have a signature of `(previousState, action) => newState`, is known as a **reducer function**, and must be *pure* and predictable.

Beyond that, Redux does not really care how you actually structure your logic inside that [reducer](#) function, as long as it obeys those basic rules. This is both a source of freedom and a source of confusion. However, there are a number of common patterns that are widely used when writing reducers, as well as a number of related topics and concepts to be aware of. As an application grows, these patterns play a crucial role in managing [reducer](#) code complexity, handling real-world data, and optimizing UI performance.

Prerequisite Concepts for Writing Reducers

Some of these concepts are already described elsewhere in the Redux documentation. Others are generic and applicable outside of Redux itself, and there are numerous existing articles that cover these concepts in detail. These concepts and techniques form the foundation of writing solid Redux [reducer](#) logic.

It is vital that these Prerequisite Concepts are **thoroughly understood** before moving on to more advanced and Redux-specific techniques. A recommended reading list is available at:

Prerequisite Concepts

It's also important to note that some of these suggestions may or may not be directly applicable based on architectural decisions in a specific application. For example, an application using Immutable.js Maps to [store](#) data would likely have its [reducer](#) logic structured at least somewhat differently than an application using plain Javascript objects. This documentation primarily assumes use of plain Javascript objects, but many of the principles would still apply if using other tools.

Reducer Concepts and Techniques

- [Basic Reducer Structure](#)
- [Splitting Reducer Logic](#)
- [Refactoring Reducers Example](#)
- [Using `combineReducers`](#)
- [Beyond `combineReducers`](#)
- [Normalizing State Shape](#)
- [Updating Normalized Data](#)
- [Reusing Reducer Logic](#)
- [Immutable Update Patterns](#)
- [Initializing State](#)

Prerequisite Reducer Concepts

As described in [Reducers](#), a Redux [reducer](#) function:

- Should have a signature of `(previousState, action) => newState`, similar to the type of function you would pass to `Array.prototype.reduce(reducer, ?initialValue)`
- Should be "pure", which means the [reducer](#):
 - Does not *perform side effects* (such as calling API's or modifying non-local objects or variables).
 - Does not *call non-pure functions* (like `Date.now` OR `Math.random`).
 - Does not *mutate* its arguments. If the [reducer](#) updates [state](#), it should not *modify* the **existing state** object in-place. Instead, it should generate a **new** object containing the necessary changes. The same approach should be used for any sub-objects within [state](#) that the [reducer](#) updates.

Note on immutability, side effects, and mutation

Mutation is discouraged because it generally breaks time-travel debugging, and React Redux's `connect` function:

- For time traveling, the Redux DevTools expect that replaying recorded actions would output a [state](#) value, but not change anything else. **Side effects like mutation or asynchronous behavior will cause time travel to alter behavior between steps, breaking the application.**
- For React Redux, `connect` checks to see if the props returned from a `mapStateToProps` function have changed in order to determine if a component needs to update. To improve performance, `connect` takes some shortcuts that rely on the [state](#) being immutable, and uses shallow reference equality checks to detect changes. This means that **changes made to objects and arrays by direct mutation will not be detected, and components will not re-render.**

Other side effects like generating unique IDs or timestamps in a [reducer](#) also make the code unpredictable and harder to debug and test.

Because of these rules, it's important that the following core concepts are fully understood before moving on to other specific techniques for organizing Redux reducers:

Redux Reducer Basics

Key concepts:

- Thinking in terms of [state](#) and [state](#) shape
- Delegating update responsibility by slice of [state](#) ([reducer composition](#))
- Higher order reducers
- Defining [reducer](#) initial [state](#)

Reading list:

- [Redux Docs: Reducers](#)
- [Redux Docs: Reducing Boilerplate](#)
- [Redux Docs: Implementing Undo History](#)
- [Redux Docs: `combineReducers`](#)
- [The Power of Higher-Order Reducers](#)
- [Stack Overflow: Store initial state and `combineReducers`](#)
- [Stack Overflow: State key names and `combineReducers`](#)

Pure Functions and Side Effects

Key Concepts:

- Side effects
- Pure functions
- How to think in terms of combining functions

Reading List:

- [The Little Idea of Functional Programming](#)
- [Understanding Programmatic Side-Effects](#)
- [Learning Functional Programming in Javascript](#)
- [An Introduction to Reasonably Pure Functional Programming](#)

Immutable Data Management

Key Concepts:

- Mutability vs immutability
- Immutably updating objects and arrays safely
- Avoiding functions and statements that mutate [state](#)

Reading List:

- [Pros and Cons of Using Immutability With React](#)
- [Immutable Data using ES6 and Beyond](#)
- [Immutable Data from Scratch](#)
- [Redux Docs: Using the Object Spread Operator](#)

Normalizing Data

Key Concepts:

- Database structure and organization
- Splitting relational/nested data up into separate tables
- Storing a single definition for a given item
- Referring to items by IDs
- Using objects keyed by item IDs as lookup tables, and arrays of IDs to track ordering
- Associating items in relationships

Reading List:

- [Database Normalization in Simple English](#)
- [Idiomatic Redux: Normalizing the State Shape](#)
- [Normalizr Documentation](#)
- [Redux Without Profanity: Normalizr](#)
- [Querying a Redux Store](#)
- [Wikipedia: Associative Entity](#)
- [Database Design: Many-to-Many](#)
- [Avoiding Accidental Complexity When Structuring Your App State](#)

Basic Reducer Structure and State Shape

Basic Reducer Structure

First and foremost, it's important to understand that your entire application really only has **one single reducer function**: the function that you've passed into `createStore` as the first argument. That one single **reducer** function ultimately needs to do several things:

- The first time the **reducer** is called, the `state` value will be `undefined`. The **reducer** needs to handle this case by supplying a default `state` value before handling the incoming **action**.
- It needs to look at the previous `state` and the dispatched **action**, and determine what kind of work needs to be done
- Assuming actual changes need to occur, it needs to create new objects and arrays with the updated data and return those
- If no changes are needed, it should return the existing `state` as-is.

The simplest possible approach to writing **reducer** logic is to put everything into a single function declaration, like this:

```
function counter(state, action) {
  if (typeof state === 'undefined') {
    state = 0 // If state is undefined, initialize it with a default value
  }

  if (action.type === 'INCREMENT') {
    return state + 1
  } else if (action.type === 'DECREMENT') {
    return state - 1
  } else {
    return state // In case an action is passed in we don't understand
  }
}
```

Notice that this simple function fulfills all the basic requirements. It returns a default value if none exists, initializing the **store**; it determines what sort of update needs to be done based on the type of the **action**, and returns new values; and it returns the previous `state` if no work needs to be done.

There are some simple tweaks that can be made to this **reducer**. First, repeated `if / else` statements quickly grow tiresome, so it's very common to use `switch` statements instead. Second, we can use ES6's default parameter values to handle the initial "no existing data" case. With those changes, the **reducer** would look like:

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

This is the basic structure that a typical Redux **reducer** function uses.

Basic State Shape

Redux encourages you to think about your application in terms of the data you need to manage. The data at any given point in time is the "*state*" of your application, and the structure and organization of that *state* is typically referred to as its "*shape*". The shape of your *state* plays a major role in how you structure your *reducer* logic.

A Redux *state* usually has a plain Javascript object as the top of the *state* tree. (It is certainly possible to have another type of data instead, such as a single number, an array, or a specialized data structure, but most libraries assume that the top-level value is a plain object.) The most common way to organize data within that top-level object is to further divide data into sub-trees, where each top-level key represents some "domain" or "slice" of related data. For example, a basic Todo app's *state* might look like:

```
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
```

In this example, `todos` and `visibilityFilter` are both top-level keys in the *state*, and each represents a "slice" of data for some particular concept.

Most applications deal with multiple types of data, which can be broadly divided into three categories:

- *Domain data*: data that the application needs to show, use, or modify (such as "all of the Todos retrieved from the server")
- *App state*: data that is specific to the application's behavior (such as "Todo #5 is currently selected", or "there is a request in progress to fetch Todos")
- *UI state*: data that represents how the UI is currently displayed (such as "The EditTodo modal dialog is currently open")

Because the *store* represents the core of your application, you should **define your *state* shape in terms of your domain data and app state, not your UI component tree**. As an example, a shape of

`state.leftPane.todoList.todos` would be a bad idea, because the idea of "todos" is central to the whole application, not just a single part of the UI. The `todos` slice should be at the top of the *state* tree instead.

There will *rarely* be a 1-to-1 correspondence between your UI tree and your *state* shape. The exception to that might be if you are explicitly tracking various aspects of UI data in your Redux *store* as well, but even then the shape of the UI data and the shape of the domain data would likely be different.

A typical app's *state* shape might look roughly like:

```
{
  domainData1 : {},
  domainData2 : {},
  appState1 : {},
  appState2 : {},
  ui : {
    uiState1 : {},
```

```
        uiState2 : {},  
    }  
}
```

Splitting Up Reducer Logic

For any meaningful application, putting *all* your update logic into a single `reducer` function is quickly going to become unmanageable. While there's no single rule for how long a function should be, it's generally agreed that functions should be relatively short and ideally only do one specific thing. Because of this, it's good programming practice to take pieces of code that are very long or do many different things, and break them into smaller pieces that are easier to understand.

Since a Redux `reducer` is *just* a function, the same concept applies. You can split some of your `reducer` logic out into another function, and call that new function from the parent function.

These new functions would typically fall into one of three categories:

1. Small utility functions containing some reusable chunk of logic that is needed in multiple places (which may or may not be actually related to the specific business logic)
2. Functions for handling a specific update case, which often need parameters other than the typical `(state, action)` pair
3. Functions which handle *all* updates for a given slice of `state`. These functions do generally have the typical `(state, action)` parameter signature

For clarity, these terms will be used to distinguish between different types of functions and different use cases:

- **`reducer`**: any function with the signature `(state, action) -> newState` (ie, any function that *could* be used as an argument to `Array.prototype.reduce`)
- **`root reducer`**: the `reducer` function that is actually passed as the first argument to `createStore` . This is the only part of the `reducer` logic that *must* have the `(state, action) -> newState` signature.
- **`slice reducer`**: a `reducer` that is being used to handle updates to one specific slice of the `state` tree, usually done by passing it to `combineReducers`
- **`case function`**: a function that is being used to handle the update logic for a specific `action`. This may actually be a `reducer` function, or it may require other parameters to do its work properly.
- **`higher-order reducer`**: a function that takes a `reducer` function as an argument, and/or returns a new `reducer` function as a result (such as `combineReducers` , OR `redux-undo`)

The term "*sub-reducer*" has also been used in various discussions to mean any function that is not the root `reducer`, although the term is not very precise. Some people may also refer to some functions as "*business logic*" (functions that relate to application-specific behavior) or "*utility functions*" (generic functions that are not application-specific).

Breaking down a complex process into smaller, more understandable parts is usually described with the term **`functional decomposition`**. This term and concept can be applied generically to any code. However, in Redux it is very common to structure `reducer` logic using approach #3, where update logic is delegated to other functions based on slice of `state`. Redux refers to this concept as **`reducer composition`**, and it is by far the most widely-used approach to structuring `reducer` logic. In fact, it's so common that Redux includes a utility function called `combineReducers()` , which specifically abstracts the process of delegating work to other `reducer` functions based on slices of `state`. However, it's important to note that it is not the *only* pattern that can be used. In fact, it's entirely possible to use all three approaches for splitting up logic into functions, and usually a good idea as well. The [Refactoring Reducers](#) section shows some examples of this in [action](#).

Refactoring Reducer Logic Using Functional Decomposition and Reducer Composition

It may be helpful to see examples of what the different types of sub-reducer functions look like and how they fit together. Let's look at a demonstration of how a large single reducer function can be refactored into a composition of several smaller functions.

Note: this example is deliberately written in a verbose style in order to illustrate the concepts and the process of refactoring, rather than perfectly concise code.

Initial Reducer

Let's say that our initial reducer looks like this:

```
const initialState = {
  visibilityFilter: 'SHOW_ALL',
  todos: []
}

function appReducer(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER': {
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    }
    case 'ADD_TODO': {
      return Object.assign({}, state, {
        todos: state.todos.concat({
          id: action.id,
          text: action.text,
          completed: false
        })
      })
    }
    case 'TOGGLE_TODO': {
      return Object.assign({}, state, {
        todos: state.todos.map(todo => {
          if (todo.id !== action.id) {
            return todo
          }

          return Object.assign({}, todo, {
            completed: !todo.completed
          })
        })
      })
    }
    case 'EDIT_TODO': {
      return Object.assign({}, state, {
        todos: state.todos.map(todo => {
          if (todo.id !== action.id) {
            return todo
          }

          return Object.assign({}, todo, {
            text: action.text
          })
        })
      })
    }
  }
}
```

```

    })
  })
}
default:
  return state
}
}

```

That function is fairly short, but already becoming overly complex. We're dealing with two different areas of concern (filtering vs managing our list of todos), the nesting is making the update logic harder to read, and it's not exactly clear what's going on everywhere.

Extracting Utility Functions

A good first step might be to break out a utility function to return a new object with updated fields. There's also a repeated pattern with trying to update a specific item in an array that we could extract to a function:

```

function updateObject(oldObject, newValues) {
  // Encapsulate the idea of passing a new object as the first parameter
  // to Object.assign to ensure we correctly copy data instead of mutating
  return Object.assign({}, oldObject, newValues)
}

function updateItemInArray(array, itemId, updateItemCallback) {
  const updatedItems = array.map(item => {
    if (item.id !== itemId) {
      // Since we only want to update one item, preserve all others as they are now
      return item
    }

    // Use the provided callback to create an updated item
    const updatedItem = updateItemCallback(item)
    return updatedItem
  })

  return updatedItems
}

function appReducer(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER': {
      return updateObject(state, { visibilityFilter: action.filter })
    }
    case 'ADD_TODO': {
      const newTodos = state.todos.concat({
        id: action.id,
        text: action.text,
        completed: false
      })

      return updateObject(state, { todos: newTodos })
    }
    case 'TOGGLE_TODO': {
      const newTodos = updateItemInArray(state.todos, action.id, todo => {
        return updateObject(todo, { completed: !todo.completed })
      })

      return updateObject(state, { todos: newTodos })
    }
    case 'EDIT_TODO': {
      const newTodos = updateItemInArray(state.todos, action.id, todo => {
        return updateObject(todo, { text: action.text })
      })
    }
  }
}

```

```

    })

    return updateObject(state, { todos: newTodos })
  }
  default:
    return state
  }
}

```

That reduced the duplication and made things a bit easier to read.

Extracting Case Reducers

Next, we can split each specific case into its own function:

```

// Omitted
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}

function setVisibilityFilter(state, action) {
  return updateObject(state, { visibilityFilter: action.filter })
}

function addTodo(state, action) {
  const newTodos = state.todos.concat({
    id: action.id,
    text: action.text,
    completed: false
  })

  return updateObject(state, { todos: newTodos })
}

function toggleTodo(state, action) {
  const newTodos = updateItemInArray(state.todos, action.id, todo => {
    return updateObject(todo, { completed: !todo.completed })
  })

  return updateObject(state, { todos: newTodos })
}

function editTodo(state, action) {
  const newTodos = updateItemInArray(state.todos, action.id, todo => {
    return updateObject(todo, { text: action.text })
  })

  return updateObject(state, { todos: newTodos })
}

function appReducer(state = initialState, action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return setVisibilityFilter(state, action)
    case 'ADD_TODO':
      return addTodo(state, action)
    case 'TOGGLE_TODO':
      return toggleTodo(state, action)
    case 'EDIT_TODO':
      return editTodo(state, action)
    default:
      return state
  }
}

```

Now it's very clear what's happening in each case. We can also start to see some patterns emerging.

Separating Data Handling by Domain

Our app `reducer` is still aware of all the different cases for our application. Let's try splitting things up so that the filter logic and the todo logic are separated:

```
// Omitted
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}

function setVisibilityFilter(visibilityState, action) {
  // Technically, we don't even care about the previous state
  return action.filter
}

function visibilityReducer(visibilityState = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return setVisibilityFilter(visibilityState, action)
    default:
      return visibilityState
  }
}

function addTodo(todosState, action) {
  const newTodos = todosState.concat([
    id: action.id,
    text: action.text,
    completed: false
  ])

  return newTodos
}

function toggleTodo(todosState, action) {
  const newTodos = updateItemInArray(todosState, action.id, todo => {
    return updateObject(todo, { completed: !todo.completed })
  })

  return newTodos
}

function editTodo(todosState, action) {
  const newTodos = updateItemInArray(todosState, action.id, todo => {
    return updateObject(todo, { text: action.text })
  })

  return newTodos
}

function todosReducer(todosState = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return addTodo(todosState, action)
    case 'TOGGLE_TODO':
      return toggleTodo(todosState, action)
    case 'EDIT_TODO':
      return editTodo(todosState, action)
    default:
      return todosState
  }
}
```

```
function appReducer(state = initialState, action) {
  return {
    todos: todosReducer(state.todos, action),
    visibilityFilter: visibilityReducer(state.visibilityFilter, action)
  }
}
```

Notice that because the two "slice of `state`" reducers are now getting only their own part of the whole `state` as arguments, they no longer need to return complex nested `state` objects, and are now simpler as a result.

Reducing Boilerplate

We're almost done. Since many people don't like switch statements, it's very common to use a function that creates a lookup table of `action` types to case functions. We'll use the `createReducer` function described in [Reducing Boilerplate](#):

```
// Omitted
function updateObject(oldObject, newValues) {}
function updateItemInArray(array, itemId, updateItemCallback) {}

function createReducer(initialState, handlers) {
  return function reducer(state = initialState, action) {
    if (handlers.hasOwnProperty(action.type)) {
      return handlers[action.type](state, action)
    } else {
      return state
    }
  }
}

// Omitted
function setVisibilityFilter(visibilityState, action) {}

const visibilityReducer = createReducer('SHOW_ALL', {
  SET_VISIBILITY_FILTER: setVisibilityFilter
})

// Omitted
function addTodo(todosState, action) {}
function toggleTodo(todosState, action) {}
function editTodo(todosState, action) {}

const todosReducer = createReducer([], {
  ADD_TODO: addTodo,
  TOGGLE_TODO: toggleTodo,
  EDIT_TODO: editTodo
})

function appReducer(state = initialState, action) {
  return {
    todos: todosReducer(state.todos, action),
    visibilityFilter: visibilityReducer(state.visibilityFilter, action)
  }
}
```

Combining Reducers by Slice

As our last step, we can now use Redux's built-in `combineReducers` utility to handle the "slice-of-`state`" logic for our top-level app `reducer`. Here's the final result:

```

// Reusable utility functions

function updateObject(oldObject, newValues) {
  // Encapsulate the idea of passing a new object as the first parameter
  // to Object.assign to ensure we correctly copy data instead of mutating
  return Object.assign({}, oldObject, newValues)
}

function updateItemInArray(array, itemId, updateItemCallback) {
  const updatedItems = array.map(item => {
    if (item.id !== itemId) {
      // Since we only want to update one item, preserve all others as they are now
      return item
    }

    // Use the provided callback to create an updated item
    const updatedItem = updateItemCallback(item)
    return updatedItem
  })

  return updatedItems
}

function createReducer(initialState, handlers) {
  return function reducer(state = initialState, action) {
    if (handlers.hasOwnProperty(action.type)) {
      return handlers[action.type](state, action)
    } else {
      return state
    }
  }
}

// Handler for a specific case ("case reducer")
function setVisibilityFilter(visibilityState, action) {
  // Technically, we don't even care about the previous state
  return action.filter
}

// Handler for an entire slice of state ("slice reducer")
const visibilityReducer = createReducer('SHOW_ALL', {
  SET_VISIBILITY_FILTER: setVisibilityFilter
})

// Case reducer
function addTodo(todosState, action) {
  const newTodos = todosState.concat({
    id: action.id,
    text: action.text,
    completed: false
  })

  return newTodos
}

// Case reducer
function toggleTodo(todosState, action) {
  const newTodos = updateItemInArray(todosState, action.id, todo => {
    return updateObject(todo, { completed: !todo.completed })
  })

  return newTodos
}

// Case reducer
function editTodo(todosState, action) {

```

```

    const newTodos = updateItemInArray(todosState, action.id, todo => {
      return updateObject(todo, { text: action.text })
    })

    return newTodos
  }

  // Slice reducer
  const todosReducer = createReducer([], {
    ADD_TODO: addTodo,
    TOGGLE_TODO: toggleTodo,
    EDIT_TODO: editTodo
  })

  // "Root reducer"
  const appReducer = combineReducers({
    visibilityFilter: visibilityReducer,
    todos: todosReducer
  })

```

We now have examples of several kinds of split-up **reducer** functions: helper utilities like `updateObject` and `createReducer`, handlers for specific cases like `setVisibilityFilter` and `addTodo`, and slice-of-state handlers like `visibilityReducer` and `todosReducer`. We also can see that `appReducer` is an example of a "root reducer".

Although the final result in this example is noticeably longer than the original version, this is primarily due to the extraction of the utility functions, the addition of comments, and some deliberate verbosity for the sake of clarity, such as separate return statements. Looking at each function individually, the amount of responsibility is now smaller, and the intent is hopefully clearer. Also, in a real application, these functions would probably then be split into separate files such as `reducerUtilities.js`, `visibilityReducer.js`, `todosReducer.js`, and `rootReducer.js`.

Using `combineReducers`

Core Concepts

The most common `state` shape for a Redux app is a plain Javascript object containing "slices" of domain-specific data at each top-level key. Similarly, the most common approach to writing `reducer` logic for that `state` shape is to have "slice `reducer`" functions, each with the same `(state, action)` signature, and each responsible for managing all updates to that specific slice of `state`. Multiple slice reducers can respond to the same `action`, independently update their own slice as needed, and the updated slices are combined into the new `state` object.

Because this pattern is so common, Redux provides the `combineReducers` utility to implement that behavior. It is an example of a *higher-order reducer*, which takes an object full of slice `reducer` functions, and returns a new `reducer` function.

There are several important ideas to be aware of when using `combineReducers` :

- First and foremost, `combineReducers` is simply **a utility function to simplify the most common use case when writing Redux reducers**. You are *not* required to use it in your own application, and it does *not* handle every possible scenario. It is entirely possible to write `reducer` logic without using it, and it is quite common to need to write custom `reducer` logic for cases that `combineReducer` does not handle. (See [Beyond combineReducers](#) for examples and suggestions.)
- While Redux itself is not opinionated about how your `state` is organized, `combineReducers` enforces several rules to help users avoid common errors. (See [combineReducers](#) for details.)
- One frequently asked question is whether Redux "calls all reducers" when dispatching an `action`. Since there really is only one root `reducer` function, the default answer is "no, it does not". However, `combineReducers` has specific behavior that *does* work that way. In order to assemble the new `state` tree, `combineReducers` will call each slice `reducer` with its current slice of `state` and the current `action`, giving the slice `reducer` a chance to respond and update its slice of `state` if needed. So, in that sense, using `combineReducers` *does* "call all reducers", or at least all of the slice reducers it is wrapping.
- You can use it at all levels of your `reducer` structure, not just to create the root `reducer`. It's very common to have multiple combined reducers in various places, which are composed together to create the root `reducer`.

Defining State Shape

There are two ways to define the initial shape and contents of your `store`'s `state`. First, the `createStore` function can take `preloadedState` as its second argument. This is primarily intended for initializing the `store` with `state` that was previously persisted elsewhere, such as the browser's `localStorage`. The other way is for the root `reducer` to return the initial `state` value when the `state` argument is `undefined`. These two approaches are described in more detail in [Initializing State](#), but there are some additional concerns to be aware of when using `combineReducers`.

`combineReducers` takes an object full of slice `reducer` functions, and creates a function that outputs a corresponding `state` object with the same keys. This means that if no preloaded `state` is provided to `createStore`, the naming of the keys in the input slice `reducer` object will define the naming of the keys in the output `state` object. The correlation between these names is not always apparent, especially when using ES6 features such as default module exports and object literal shorthands.

Here's an example of how use of ES6 object literal shorthand with `combineReducers` can define the `state` shape:

```
// reducers.js
export default theDefaultReducer = (state = 0, action) => state

export const firstNamedReducer = (state = 1, action) => state

export const secondNamedReducer = (state = 2, action) => state

// rootReducer.js
import { combineReducers, createStore } from 'redux'

import theDefaultReducer, {
  firstNamedReducer,
  secondNamedReducer
} from './reducers'

// Use ES6 object literal shorthand syntax to define the object shape
const rootReducer = combineReducers({
  theDefaultReducer,
  firstNamedReducer,
  secondNamedReducer
})

const store = createStore(rootReducer)
console.log(store.getState())
// {theDefaultReducer : 0, firstNamedReducer : 1, secondNamedReducer : 2}
```

Notice that because we used the ES6 shorthand for defining an object literal, the key names in the resulting `state` are the same as the variable names from the imports. This may not always be the desired behavior, and is often a cause of confusion for those who aren't as familiar with ES6 syntax.

Also, the resulting names are a bit odd. It's generally not a good practice to actually include words like `"reducer"` in your `state` key names - the keys should simply reflect the domain or type of data they hold. This means we should either explicitly specify the names of the keys in the slice `reducer` object to define the keys in the output `state` object, or carefully rename the variables for the imported slice reducers to set up the keys when using the shorthand object literal syntax.

A better usage might look like:

```
import { combineReducers, createStore } from 'redux'

// Rename the default import to whatever name we want. We can also rename a named import.
import defaultState, {
  firstNamedReducer,
  secondNamedReducer as secondState
} from './reducers'

const rootReducer = combineReducers({
  defaultState, // key name same as the carefully renamed default export
  firstState: firstNamedReducer, // specific key name instead of the variable name
  secondState // key name same as the carefully renamed named export
})

const reducerInitializedStore = createStore(rootReducer)
console.log(reducerInitializedStore.getState())
// {defaultState : 0, firstState : 1, secondState : 2}
```

This `state` shape better reflects the data involved, because we took care to set up the keys we passed to `combineReducers`.

Beyond `combineReducers`

The `combineReducers` utility included with Redux is very useful, but is deliberately limited to handle a single common use case: updating a [state](#) tree that is a plain Javascript object, by delegating the work of updating each slice of [state](#) to a specific slice [reducer](#). It does *not* handle other use cases, such as a [state](#) tree made up of Immutable.js Maps, trying to pass other portions of the [state](#) tree as an additional argument to a slice [reducer](#), or performing "ordering" of slice [reducer](#) calls. It also does not care how a given slice [reducer](#) does its work.

The common question, then, is "How can I use `combineReducers` to handle these other use cases?". The answer to that is simply: "you don't - you probably need to use something else". **Once you go past the core use case for `combineReducers`, it's time to use more "custom" [reducer](#) logic**, whether it be specific logic for a one-off use case, or a reusable function that could be widely shared. Here's some suggestions for dealing with a couple of these typical use cases, but feel free to come up with your own approaches.

Using slice reducers with Immutable.js objects

Since `combineReducers` currently only works with plain Javascript objects, an application that uses an Immutable.js Map object for the top of its [state](#) tree could not use `combineReducers` to manage that Map. Since many developers do use Immutable.js, there are a number of published utilities that provide equivalent functionality, such as [redux-immutable](#). This package provides its own implementation of `combineReducers` that knows how to iterate over an Immutable Map instead of a plain Javascript object.

Sharing data between slice reducers

Similarly, if `sliceReducerA` happens to need some data from `sliceReducerB`'s slice of [state](#) in order to handle a particular [action](#), or `sliceReducerB` happens to need the entire [state](#) as an argument, `combineReducers` does not handle that itself. This could be resolved by writing a custom function that knows to pass the needed data as an additional argument in those specific cases, such as:

```
function combinedReducer(state, action) {
  switch (action.type) {
    case 'A_TYPICAL_ACTION': {
      return {
        a: sliceReducerA(state.a, action),
        b: sliceReducerB(state.b, action)
      }
    }
    case 'SOME_SPECIAL_ACTION': {
      return {
        // specifically pass state.b as an additional argument
        a: sliceReducerA(state.a, action, state.b),
        b: sliceReducerB(state.b, action)
      }
    }
    case 'ANOTHER_SPECIAL_ACTION': {
      return {
        a: sliceReducerA(state.a, action),
        // specifically pass the entire state as an additional argument
        b: sliceReducerB(state.b, action, state)
      }
    }
    default:
```

```

    return state
  }
}

```

Another alternative to the "shared-slice updates" issue would be to simply put more data into the [action](#). This is easily accomplished using thunk functions or a similar approach, per this example:

```

function someSpecialActionCreator() {
  return (dispatch, getState) => {
    const state = getState()
    const dataFromB = selectImportantDataFromB(state)

    dispatch({
      type: 'SOME_SPECIAL_ACTION',
      payload: {
        dataFromB
      }
    })
  }
}

```

Because the data from B's slice is already in the [action](#), the parent [reducer](#) doesn't have to do anything special to make that data available to `sliceReducerA`.

A third approach would be to use the [reducer](#) generated by `combineReducers` to handle the "simple" cases where each slice [reducer](#) can update itself independently, but also use another [reducer](#) to handle the "special" cases where data needs to be shared across slices. Then, a wrapping function could call both of those reducers in turn to generate the final result:

```

const combinedReducer = combineReducers({
  a: sliceReducerA,
  b: sliceReducerB
})

function crossSliceReducer(state, action) {
  switch (action.type) {
    case 'SOME_SPECIAL_ACTION': {
      return {
        // specifically pass state.b as an additional argument
        a: handleSpecialCaseForA(state.a, action, state.b),
        b: sliceReducerB(state.b, action)
      }
    }
    default:
      return state
  }
}

function rootReducer(state, action) {
  const intermediateState = combinedReducer(state, action)
  const finalState = crossSliceReducer(intermediateState, action)
  return finalState
}

```

As it turns out, there's a useful utility called [reduce-reducers](#) that can make that process easier. It simply takes multiple reducers and runs `reduce()` on them, passing the intermediate [state](#) values to the next [reducer](#) in line:

```

// Same as the "manual" rootReducer above
const rootReducer = reduceReducers(combinedReducers, crossSliceReducer)

```

Note that if you use `reduceReducers`, you should make sure that the first `reducer` in the list is able to define the initial `state`, since the later reducers will generally assume that the entire `state` already exists and not try to provide defaults.

Further Suggestions

Again, it's important to understand that Redux reducers are *just* functions. While `combineReducers` is useful, it's just one tool in the toolbox. Functions can contain conditional logic other than switch statements, functions can be composed to wrap each other, and functions can call other functions. Maybe you need one of your slice reducers to be able to reset its `state`, and to only respond to specific actions overall. You could do:

```
const undoableFilteredSliceA = compose(
  undoReducer,
  filterReducer('ACTION_1', 'ACTION_2'),
  sliceReducerA
)
const rootReducer = combineReducers({
  a: undoableFilteredSliceA,
  b: normalSliceReducerB
})
```

Note that `combineReducers` doesn't know or care that there's anything special about the `reducer` function that's responsible for managing `a`. We didn't need to modify `combineReducers` to specifically know how to undo things - we just built up the pieces we needed into a new composed function.

Also, while `combineReducers` is the one `reducer` utility function that's built into Redux, there's a wide variety of third-party `reducer` utilities that have published for reuse. The [Redux Addons Catalog](#) lists many of the third-party utilities that are available. Or, if none of the published utilities solve your use case, you can always write a function yourself that does just exactly what you need.

Normalizing State Shape

Many applications deal with data that is nested or relational in nature. For example, a blog editor could have many Posts, each Post could have many Comments, and both Posts and Comments would be written by a User. Data for this kind of application might look like:

```
const blogPosts = [
  {
    id: 'post1',
    author: { username: 'user1', name: 'User 1' },
    body: '.....',
    comments: [
      {
        id: 'comment1',
        author: { username: 'user2', name: 'User 2' },
        comment: '.....'
      },
      {
        id: 'comment2',
        author: { username: 'user3', name: 'User 3' },
        comment: '.....'
      }
    ]
  },
  {
    id: 'post2',
    author: { username: 'user2', name: 'User 2' },
    body: '.....',
    comments: [
      {
        id: 'comment3',
        author: { username: 'user3', name: 'User 3' },
        comment: '.....'
      },
      {
        id: 'comment4',
        author: { username: 'user1', name: 'User 1' },
        comment: '.....'
      },
      {
        id: 'comment5',
        author: { username: 'user3', name: 'User 3' },
        comment: '.....'
      }
    ]
  }
]
// and repeat many times
]
```

Notice that the structure of the data is a bit complex, and some of the data is repeated. This is a concern for several reasons:

- When a piece of data is duplicated in several places, it becomes harder to make sure that it is updated appropriately.
- Nested data means that the corresponding [reducer](#) logic has to be more nested and therefore more complex. In particular, trying to update a deeply nested field can become very ugly very fast.
- Since immutable data updates require all ancestors in the [state](#) tree to be copied and updated as well, and new object references will cause connected UI components to re-render, an update to a deeply nested data

object could force totally unrelated UI components to re-render even if the data they're displaying hasn't actually changed.

Because of this, the recommended approach to managing relational or nested data in a Redux [store](#) is to treat a portion of your [store](#) as if it were a database, and keep that data in a *normalized* form.

Designing a Normalized State

The basic concepts of normalizing data are:

- Each type of data gets its own "table" in the [state](#).
- Each "data table" should [store](#) the individual items in an object, with the IDs of the items as keys and the items themselves as the values.
- Any references to individual items should be done by storing the item's ID.
- Arrays of IDs should be used to indicate ordering.

An example of a normalized [state](#) structure for the blog example above might look like:

```
{
  posts : {
    byId : {
      "post1" : {
        id : "post1",
        author : "user1",
        body : ".....",
        comments : ["comment1", "comment2"]
      },
      "post2" : {
        id : "post2",
        author : "user2",
        body : ".....",
        comments : ["comment3", "comment4", "comment5"]
      }
    },
    allIds : ["post1", "post2"]
  },
  comments : {
    byId : {
      "comment1" : {
        id : "comment1",
        author : "user2",
        comment : ".....",
      },
      "comment2" : {
        id : "comment2",
        author : "user3",
        comment : ".....",
      },
      "comment3" : {
        id : "comment3",
        author : "user3",
        comment : ".....",
      },
      "comment4" : {
        id : "comment4",
        author : "user1",
        comment : ".....",
      },
      "comment5" : {
        id : "comment5",
        author : "user3",
      }
    }
  }
}
```



```

        comment : "....",
      },
    },
    allIds : ["comment1", "comment2", "comment3", "comment4", "comment5"]
  },
  users : {
    byId : {
      "user1" : {
        username : "user1",
        name : "User 1",
      },
      "user2" : {
        username : "user2",
        name : "User 2",
      },
      "user3" : {
        username : "user3",
        name : "User 3",
      }
    },
    allIds : ["user1", "user2", "user3"]
  }
}

```

This **state** structure is much flatter overall. Compared to the original nested format, this is an improvement in several ways:

- Because each item is only defined in one place, we don't have to try to make changes in multiple places if that item is updated.
- The **reducer** logic doesn't have to deal with deep levels of nesting, so it will probably be much simpler.
- The logic for retrieving or updating a given item is now fairly simple and consistent. Given an item's type and its ID, we can directly look it up in a couple simple steps, without having to dig through other objects to find it.
- Since each data type is separated, an update like changing the text of a comment would only require new copies of the "comments > byId > comment" portion of the tree. This will generally mean fewer portions of the UI that need to update because their data has changed. In contrast, updating a comment in the original nested shape would have required updating the comment object, the parent post object, the array of all post objects, and likely have caused *all* of the Post components and Comment components in the UI to re-render themselves.

Note that a normalized **state** structure generally implies that more components are connected and each component is responsible for looking up its own data, as opposed to a few connected components looking up large amounts of data and passing all that data downwards. As it turns out, having connected parent components simply pass item IDs to connected children is a good pattern for optimizing UI performance in a React Redux application, so keeping **state** normalized plays a key role in improving performance.

Organizing Normalized Data in State

A typical application will likely have a mixture of relational data and non-relational data. While there is no single rule for exactly how those different types of data should be organized, one common pattern is to put the relational "tables" under a common parent key, such as "entities". A **state** structure using this approach might look like:

```

{
  simpleDomainData1: {...},
  simpleDomainData2: {...},
  entities : {
    entityType1 : {...},

```

```

    entityType2 : {...}
  },
  ui : {
    uiSection1 : {...},
    uiSection2 : {...}
  }
}

```

This could be expanded in a number of ways. For example, an application that does a lot of editing of entities might want to keep two sets of "tables" in the [state](#), one for the "current" item values and one for the "work-in-progress" item values. When an item is edited, its values could be copied into the "work-in-progress" section, and any actions that update it would be applied to the "work-in-progress" copy, allowing the editing form to be controlled by that set of data while another part of the UI still refers to the original version. "Resetting" the edit form would simply require removing the item from the "work-in-progress" section and re-copying the original data from "current" to "work-in-progress", while "applying" the edits would involve copying the values from the "work-in-progress" section to the "current" section.

Relationships and Tables

Because we're treating a portion of our Redux [store](#) as a "database", many of the principles of database design also apply here as well. For example, if we have a many-to-many relationship, we can model that using an intermediate table that stores the IDs of the corresponding items (often known as a "join table" or an "associative table"). For consistency, we would probably also want to use the same `byId` and `allIds` approach that we used for the actual item tables, like this:

```

{
  entities: {
    authors : { byId : {}, allIds : [] },
    books : { byId : {}, allIds : [] },
    authorBook : {
      byId : {
        1 : {
          id : 1,
          authorId : 5,
          bookId : 22
        },
        2 : {
          id : 2,
          authorId : 5,
          bookId : 15,
        },
        3 : {
          id : 3,
          authorId : 42,
          bookId : 12
        }
      },
      allIds : [1, 2, 3]
    }
  }
}

```

Operations like "Look up all books by this author", can then be accomplished easily with a single loop over the join table. Given the typical amounts of data in a client application and the speed of Javascript engines, this kind of operation is likely to have sufficiently fast performance for most use cases.

Normalizing Nested Data

Because APIs frequently send back data in a nested form, that data needs to be transformed into a normalized shape before it can be included in the [state](#) tree. The [Normalizr](#) library is usually used for this task. You can define schema types and relations, feed the schema and the response data to Normalizr, and it will output a normalized transformation of the response. That output can then be included in an [action](#) and used to update the [store](#). See the Normalizr documentation for more details on its usage.

Managing Normalized Data

As mentioned in [Normalizing State Shape](#), the Normalizr library is frequently used to transform nested response data into a normalized shape suitable for integration into the [store](#). However, that doesn't address the issue of executing further updates to that normalized data as it's being used elsewhere in the application. There are a variety of different approaches that you can use, based on your own preference. We'll use the example of adding a new Comment to a Post.

Standard Approaches

Simple Merging

One approach is to merge the contents of the [action](#) into the existing [state](#). In this case, we need to do a deep recursive merge, not just a shallow copy. The Lodash `merge` function can handle this for us:

```
import merge from 'lodash/merge'

function commentsById(state = {}, action) {
  switch (action.type) {
    default: {
      if (action.entities && action.entities.comments) {
        return merge({}, state, action.entities.comments.byId)
      }
      return state
    }
  }
}
```

This requires the least amount of work on the [reducer](#) side, but does require that the [action](#) creator potentially do a fair amount of work to organize the data into the correct shape before the [action](#) is dispatched. It also doesn't handle trying to delete an item.

Slice Reducer Composition

If we have a nested tree of slice reducers, each slice [reducer](#) will need to know how to respond to this [action](#) appropriately. We will need to include all the relevant data in the [action](#). We need to update the correct Post object with the comment's ID, create a new Comment object using that ID as a key, and include the Comment's ID in the list of all Comment IDs. Here's how the pieces for this might fit together:

```
// actions.js
function addComment(postId, commentText) {
  // Generate a unique ID for this comment
  const commentId = generateId('comment')

  return {
    type: 'ADD_COMMENT',
    payload: {
      postId,
      commentId,
      commentText
    }
  }
}
```

```

// reducers/posts.js
function addComment(state, action) {
  const { payload } = action
  const { postId, commentId } = payload

  // Look up the correct post, to simplify the rest of the code
  const post = state[postId]

  return {
    ...state,
    // Update our Post object with a new "comments" array
    [postId]: {
      ...post,
      comments: post.comments.concat(commentId)
    }
  }
}

function postsById(state = {}, action) {
  switch (action.type) {
    case 'ADD_COMMENT':
      return addComment(state, action)
    default:
      return state
  }
}

function allPosts(state = [], action) {
  // omitted - no work to be done for this example
}

const postsReducer = combineReducers({
  byId: postsById,
  allIds: allPosts
})

// reducers/comments.js
function addCommentEntry(state, action) {
  const { payload } = action
  const { commentId, commentText } = payload

  // Create our new Comment object
  const comment = { id: commentId, text: commentText }

  // Insert the new Comment object into the updated lookup table
  return {
    ...state,
    [commentId]: comment
  }
}

function commentsById(state = {}, action) {
  switch (action.type) {
    case 'ADD_COMMENT':
      return addCommentEntry(state, action)
    default:
      return state
  }
}

function addCommentId(state, action) {
  const { payload } = action
  const { commentId } = payload
  // Just append the new Comment's ID to the list of all IDs
  return state.concat(commentId)
}

```

```

}

function allComments(state = [], action) {
  switch (action.type) {
    case 'ADD_COMMENT':
      return addCommentId(state, action)
    default:
      return state
  }
}

const commentsReducer = combineReducers({
  byId: commentsById,
  allIds: allComments
})

```

The example is a bit long, because it's showing how all the different slice reducers and case reducers fit together. Note the delegation involved here. The `postsById` slice reducer delegates the work for this case to `addComment`, which inserts the new Comment's ID into the correct Post item. Meanwhile, both the `commentsById` and `allComments` slice reducers have their own case reducers, which update the Comments lookup table and list of all Comment IDs appropriately.

Other Approaches

Task-Based Updates

Since reducers are just functions, there's an infinite number of ways to split up this logic. While using slice reducers is the most common, it's also possible to organize behavior in a more task-oriented structure. Because this will often involve more nested updates, you may want to use an immutable update utility library like [dot-prop-immutable](#) or [object-path-immutable](#) to simplify the update statements. Here's an example of what that might look like:

```

import posts from './postsReducer';
import comments from './commentsReducer';
import dotProp from 'dot-prop-immutable';
import {combineReducers} from 'redux';
import reduceReducers from 'reduce-reducers';

const combinedReducer = combineReducers({
  posts,
  comments
});

function addComment(state, action) {
  const {payload} = action;
  const {postId, commentId, commentText} = payload;

  // State here is the entire combined state
  const updatedWithPostState = dotProp.set(
    state,
    `posts.byId.${postId}.comments`,
    comments => comments.concat(commentId)
  );

  const updatedWithCommentsTable = dotProp.set(
    updatedWithPostState,
    `comments.byId.${commentId}`,
    {id : commentId, text : commentText}
  );
}

```

```

    );

    const updatedWithCommentsList = dotProp.set(
      updatedWithCommentsTable,
      `comments.allIds`,
      allIds => allIds.concat(commentId);
    );

    return updatedWithCommentsList;
  }

  const featureReducers = createReducer({}, {
    ADD_COMMENT : addComment,
  });

  const rootReducer = reduceReducers(
    combinedReducer,
    featureReducers
  );

```

This approach makes it very clear what's happening for the `"ADD_COMMENTS"` case, but it does require nested updating logic, and some specific knowledge of the [state](#) tree shape. Depending on how you want to compose your [reducer](#) logic, this may or may not be desired.

Redux-ORM

The [Redux-ORM](#) library provides a very useful abstraction layer for managing normalized data in a [Redux store](#). It allows you to declare Model classes and define relations between them. It can then generate the empty "tables" for your data types, act as a specialized selector tool for looking up the data, and perform immutable updates on that data.

There's a couple ways Redux-ORM can be used to perform updates. First, the Redux-ORM docs suggest defining [reducer](#) functions on each Model subclass, then including the auto-generated combined [reducer](#) function into your [store](#):

```

// models.js
import { Model, fk, attr, ORM } from 'redux-orm'

export class Post extends Model {
  static get fields() {
    return {
      id: attr(),
      name: attr()
    }
  }

  static reducer(action, Post, session) {
    switch (action.type) {
      case 'CREATE_POST': {
        Post.create(action.payload)
        break
      }
    }
  }
}

Post.modelName = 'Post'

export class Comment extends Model {
  static get fields() {
    return {
      id: attr(),

```

```

    text: attr(),
    // Define a foreign key relation - one Post can have many Comments
    postId: fk({
      to: 'Post', // must be the same as Post.modelName
      as: 'post', // name for accessor (comment.post)
      relatedName: 'comments' // name for backward accessor (post.comments)
    })
  }
}

static reducer(action, Comment, session) {
  switch (action.type) {
    case 'ADD_COMMENT': {
      Comment.create(action.payload)
      break
    }
  }
}
}
Comment.modelName = 'Comment'

// Create an ORM instance and hook up the Post and Comment models
export const orm = new ORM()
orm.register(Post, Comment)

// main.js
import { createStore, combineReducers } from 'redux'
import { createReducer } from 'redux-orm'
import { orm } from './models'

const rootReducer = combineReducers({
  // Insert the auto-generated Redux-ORM reducer. This will
  // initialize our model "tables", and hook up the reducer
  // logic we defined on each Model subclass
  entities: createReducer(orm)
})

// Dispatch an action to create a Post instance
store.dispatch({
  type: 'CREATE_POST',
  payload: {
    id: 1,
    name: 'Test Post Please Ignore'
  }
})

// Dispatch an action to create a Comment instance as a child of that Post
store.dispatch({
  type: 'ADD_COMMENT',
  payload: {
    id: 123,
    text: 'This is a comment',
    postId: 1
  }
})

```

The Redux-ORM library maintains relationships between models for you. Updates are by default applied immutably, simplifying the update process.

Another variation on this is to use Redux-ORM as an abstraction layer within a single case [reducer](#):

```

import { orm } from './models'

// Assume this case reducer is being used in our "entities" slice reducer,

```



```
// and we do not have reducers defined on our Redux-ORM Model subclasses
function addComment(entitiesState, action) {
  // Start an immutable session
  const session = orm.session(entitiesState)

  session.Comment.create(action.payload)

  // The internal state reference has now changed
  return session.state
}
```

By using the session interface you can now use relationship accessors to directly access referenced models:

```
const session = orm.session(store.getState().entities)
const comment = session.Comment.first() // Comment instance
const { post } = comment // Post instance
post.comments.filter(c => c.text === 'This is a comment').count() // 1
```

Overall, Redux-ORM provides a very useful set of abstractions for defining relations between data types, creating the "tables" in our [state](#), retrieving and denormalizing relational data, and applying immutable updates to relational data.

Reusing Reducer Logic

As an application grows, common patterns in [reducer](#) logic will start to emerge. You may find several parts of your [reducer](#) logic doing the same kinds of work for different types of data, and want to reduce duplication by reusing the same common logic for each data type. Or, you may want to have multiple "instances" of a certain type of data being handled in the [store](#). However, the global structure of a Redux [store](#) comes with some trade-offs: it makes it easy to track the overall [state](#) of an application, but can also make it harder to "target" actions that need to update a specific piece of [state](#), particularly if you are using `combineReducers`.

As an example, let's say that we want to track multiple counters in our application, named A, B, and C. We define our initial `counter` [reducer](#), and we use `combineReducers` to set up our [state](#):

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

const rootReducer = combineReducers({
  counterA: counter,
  counterB: counter,
  counterC: counter
})
```

Unfortunately, this setup has a problem. Because `combineReducers` will call each slice [reducer](#) with the same [action](#), dispatching `{type: 'INCREMENT'}` will actually cause *all three* counter values to be incremented, not just one of them. We need some way to wrap the `counter` logic so that we can ensure that only the counter we care about is updated.

Customizing Behavior with Higher-Order Reducers

As defined in [Splitting Reducer Logic](#), a *higher-order reducer* is a function that takes a [reducer](#) function as an argument, and/or returns a new [reducer](#) function as a result. It can also be viewed as a "[reducer](#) factory".

`combineReducers` is one example of a higher-order [reducer](#). We can use this pattern to create specialized versions of our own [reducer](#) functions, with each version only responding to specific actions.

The two most common ways to specialize a [reducer](#) are to generate new [action](#) constants with a given prefix or suffix, or to attach additional info inside the [action](#) object. Here's what those might look like:

```
function createCounterWithNamedType(counterName = '') {
  return function counter(state = 0, action) {
    switch (action.type) {
      case `INCREMENT_${counterName}`:
        return state + 1
      case `DECREMENT_${counterName}`:
        return state - 1
      default:
        return state
    }
  }
}
```

```

    }
  }
}

function createCounterWithNameData(counterName = '') {
  return function counter(state = 0, action) {
    const { name } = action
    if (name !== counterName) return state

    switch (action.type) {
      case 'INCREMENT':
        return state + 1
      case 'DECREMENT':
        return state - 1
      default:
        return state
    }
  }
}

```

We should now be able to use either of these to generate our specialized counter reducers, and then dispatch actions that will affect the portion of the `state` that we care about:

```

const rootReducer = combineReducers({
  counterA: createCounterWithNamedType('A'),
  counterB: createCounterWithNamedType('B'),
  counterC: createCounterWithNamedType('C')
})

store.dispatch({ type: 'INCREMENT_B' })
console.log(store.getState())
// {counterA : 0, counterB : 1, counterC : 0}

```

We could also vary the approach somewhat, and create a more generic higher-order `reducer` that accepts both a given `reducer` function and a name or identifier:

```

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}

function createNamedWrapperReducer(reducerFunction, reducerName) {
  return (state, action) => {
    const { name } = action
    const isInitializationCall = state === undefined
    if (name !== reducerName && !isInitializationCall) return state

    return reducerFunction(state, action)
  }
}

const rootReducer = combineReducers({
  counterA: createNamedWrapperReducer(counter, 'A'),
  counterB: createNamedWrapperReducer(counter, 'B'),
  counterC: createNamedWrapperReducer(counter, 'C')
})

```

You could even go as far as to make a generic filtering higher-order [reducer](#):

```
function createFilteredReducer(reducerFunction, reducerPredicate) {
  return (state, action) => {
    const isInitializationCall = state === undefined;
    const shouldRunWrappedReducer = reducerPredicate(action) || isInitializationCall;
    return shouldRunWrappedReducer ? reducerFunction(state, action) : state;
  }
}

const rootReducer = combineReducers({
  // check for suffixed strings
  counterA : createFilteredReducer(counter, action => action.type.endsWith('_A')),
  // check for extra data in the action
  counterB : createFilteredReducer(counter, action => action.name === 'B'),
  // respond to all 'INCREMENT' actions, but never 'DECREMENT'
  counterC : createFilteredReducer(counter, action => action.type === 'INCREMENT')
});
```

These basic patterns allow you to do things like having multiple instances of a smart connected component within the UI, or reuse common logic for generic capabilities such as pagination or sorting.

In addition to generating reducers this way, you might also want to generate [action](#) creators using the same approach, and could generate them both at the same time with helper functions. See [Action/Reducer Generators](#) and [Reducers](#) libraries for [action/reducer](#) utilities.

Immutable Update Patterns

The articles listed in [Prerequisite Concepts#Immutable Data Management](#) give a number of good examples for how to perform basic update operations immutably, such as updating a field in an object or adding an item to the end of an array. However, reducers will often need to use those basic operations in combination to perform more complicated tasks. Here are some examples for some of the more common tasks you might have to implement.

Updating Nested Objects

The key to updating nested data is **that every level of nesting must be copied and updated appropriately**. This is often a difficult concept for those learning Redux, and there are some specific problems that frequently occur when trying to update nested objects. These lead to accidental direct mutation, and should be avoided.

Common Mistake #1: New variables that point to the same objects

Defining a new variable does *not* create a new actual object - it only creates another reference to the same object. An example of this error would be:

```
function updateNestedState(state, action) {
  let nestedState = state.nestedState
  // ERROR: this directly modifies the existing object reference - don't do this!
  nestedState.nestedField = action.data

  return {
    ...state,
    nestedState
  }
}
```

This function does correctly return a shallow copy of the top-level `state` object, but because the `nestedState` variable was still pointing at the existing object, the `state` was directly mutated.

Common Mistake #2: Only making a shallow copy of one level

Another common version of this error looks like this:

```
function updateNestedState(state, action) {
  // Problem: this only does a shallow copy!
  let newState = { ...state }

  // ERROR: nestedState is still the same object!
  newState.nestedState.nestedField = action.data

  return newState
}
```

Doing a shallow copy of the top level is *not* sufficient - the `nestedState` object should be copied as well.

Correct Approach: Copying All Levels of Nested Data

Unfortunately, the process of correctly applying immutable updates to deeply nested `state` can easily become verbose and hard to read. Here's what an example of updating `state.first.second[someId].fourth` might look like:

```
function updateVeryNestedField(state, action) {
  return {
    ...state,
    first: {
      ...state.first,
      second: {
        ...state.first.second,
        [action.someId]: {
          ...state.first.second[action.someId],
          fourth: action.someValue
        }
      }
    }
  }
}
```

Obviously, each layer of nesting makes this harder to read, and gives more chances to make mistakes. This is one of several reasons why you are encouraged to keep your `state` flattened, and compose reducers as much as possible.

Inserting and Removing Items in Arrays

Normally, a Javascript array's contents are modified using mutative functions like `push`, `unshift`, and `splice`. Since we don't want to mutate `state` directly in reducers, those should normally be avoided. Because of that, you might see "insert" or "remove" behavior written like this:

```
function insertItem(array, action) {
  return [
    ...array.slice(0, action.index),
    action.item,
    ...array.slice(action.index)
  ]
}

function removeItem(array, action) {
  return [...array.slice(0, action.index), ...array.slice(action.index + 1)]
}
```

However, remember that the key is that the *original in-memory reference* is not modified. **As long as we make a copy first, we can safely mutate the copy.** Note that this is true for both arrays and objects, but nested values still must be updated using the same rules.

This means that we could also write the insert and remove functions like this:

```
function insertItem(array, action) {
  let newArray = array.slice()
  newArray.splice(action.index, 0, action.item)
  return newArray
}

function removeItem(array, action) {
  let newArray = array.slice()
  newArray.splice(action.index, 1)
  return newArray
}
```

The remove function could also be implemented as:

```
function removeItem(array, action) {
  return array.filter((item, index) => index !== action.index)
}
```

Updating an Item in an Array

Updating one item in an array can be accomplished by using `Array.map`, returning a new value for the item we want to update, and returning the existing values for all other items:

```
function updateObjectInArray(array, action) {
  return array.map((item, index) => {
    if (index !== action.index) {
      // This isn't the item we care about - keep it as-is
      return item
    }

    // Otherwise, this is the one we want - return an updated value
    return {
      ...item,
      ...action.item
    }
  })
}
```

Immutable Update Utility Libraries

Because writing immutable update code can become tedious, there are a number of utility libraries that try to abstract out the process. These libraries vary in APIs and usage, but all try to provide a shorter and more succinct way of writing these updates. For example, [Immer](#) makes immutable updates a simple function and plain JavaScript objects:

```
var usersState = [{ name: 'John Doe', address: { city: 'London' } }]
var newState = immer.produce(usersState, draftState => {
  draftState[0].name = 'Jon Doe'
  draftState[0].address.city = 'Paris'
  //nested update similar to mutable way
})
```

Some, like [dot-prop-immutable](#), take string paths for commands:

```
state = dotProp.set(state, `todos.${index}.complete`, true)
```

Others, like [immutability-helper](#) (a fork of the now-deprecated React Immutability Helpers addon), use nested values and helper functions:

```
var collection = [1, 2, { a: [12, 17, 15] }]
var newCollection = update(collection, {
  2: { a: { $splice: [[1, 1, 13, 14]] } }
})
```

They can provide a useful alternative to writing manual immutable update logic.

A list of many immutable update utilities can be found in the [Immutable Data#Immutable Update Utilities](#) section of the [Redux Addons Catalog](#).

Simplifying Immutable Updates with Redux Starter Kit

Our [Redux Starter Kit](#) package includes a `createReducer` utility that uses Immer internally. Because of this, you can write reducers that appear to "mutate" [state](#), but the updates are actually applied immutably.

This allows immutable update logic to be written in a much simpler way. Here's what the [nested data example](#) might look like using `createReducer`:

```
import { createReducer } from 'redux-starter-kit'

const initialState = {
  first: {
    second: {
      id1: { fourth: 'a' },
      id2: { fourth: 'b' }
    }
  }
}

const reducer = createReducer(initialState, {
  UPDATE_ITEM: (state, action) => {
    state.first.second[action.someId].fourth = action.someValue
  }
})
```

This is clearly *much* shorter and easier to read. However, **this *only* works correctly if you are using the "magic" `createReducer` function from Redux Starter Kit** that wraps this [reducer](#) in Immer's [produce](#) function. **If this [reducer](#) is used without Immer, it will actually mutate the [state](#)!** It's also not obvious just by looking at the code that this function is actually safe and updates the [state](#) immutably. Please make sure you understand the concepts of immutable updates fully. If you do use this, it may help to add some comments to your code that explain your reducers are using Redux Starter Kit and Immer.

Initializing State

There are two main ways to initialize `state` for your application. The `createStore` method can accept an optional `preloadedState` value as its second argument. Reducers can also specify an initial value by looking for an incoming `state` argument that is `undefined`, and returning the value they'd like to use as a default. This can either be done with an explicit check inside the `reducer`, or by using the ES6 default argument value syntax: `function myReducer(state = someDefaultValue, action) {`

It's not always immediately clear how these two approaches interact. Fortunately, the process does follow some predictable rules. Here's how the pieces fit together.

Summary

Without `combineReducers()` or similar manual code, `preloadedState` always wins over `state = ...` in the `reducer` because the `state` passed to the `reducer` is `preloadedState` and is *not* `undefined`, so the ES6 argument syntax doesn't apply.

With `combineReducers()` the behavior is more nuanced. Those reducers whose `state` is specified in `preloadedState` will receive that `state`. Other reducers will receive `undefined` and because of that will fall back to the `state = ...` default argument they specify.

In general, `preloadedState` wins over the `state` specified by the `reducer`. This lets reducers specify initial data that makes sense to them as default arguments, but also allows loading existing data (fully or partially) when you're hydrating the `store` from some persistent storage or the server.

Note: Reducers whose initial `state` is populated using `preloadedState` will still need to provide a default value to handle when passed a `state` of `undefined`. All reducers are passed `undefined` on initialization, so they should be written such that when given `undefined`, some value should be returned. This can be any non-`undefined` value; there's no need to duplicate the section of `preloadedState` here as the default.

In Depth

Single Simple Reducer

First let's consider a case where you have a single `reducer`. Say you don't use `combineReducers()`.

Then your `reducer` might look like this:

```
function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

Now let's say you create a `store` with it.

```
import { createStore } from 'redux'
const store = createStore(counter)
console.log(store.getState()) // 0
```

The initial `state` is zero. Why? Because the second argument to `createStore` was `undefined`. This is the `state` passed to your `reducer` the first time. When Redux initializes it dispatches a "dummy" `action` to fill the `state`. So your `counter` `reducer` was called with `state` equal to `undefined`. **This is exactly the case that "activates" the default argument.** Therefore, `state` is now `0` as per the default `state` value (`state = 0`). This `state (0)` will be returned.

Let's consider a different scenario:

```
import { createStore } from 'redux'
const store = createStore(counter, 42)
console.log(store.getState()) // 42
```

Why is it `42`, and not `0`, this time? Because `createStore` was called with `42` as the second argument. This argument becomes the `state` passed to your `reducer` along with the dummy `action`. **This time, `state` is not undefined (it's `42`!), so ES6 default argument syntax has no effect.** The `state` is `42`, and `42` is returned from the `reducer`.

Combined Reducers

Now let's consider a case where you use `combineReducers()`.

You have two reducers:

```
function a(state = 'lol', action) {
  return state
}

function b(state = 'wat', action) {
  return state
}
```

The `reducer` generated by `combineReducers({ a, b })` looks like this:

```
// const combined = combineReducers({ a, b })
function combined(state = {}, action) {
  return {
    a: a(state.a, action),
    b: b(state.b, action)
  }
}
```

If we call `createStore` without the `preloadedState`, it's going to initialize the `state` to `{}`. Therefore, `state.a` and `state.b` will be `undefined` by the time it calls `a` and `b` reducers. **Both `a` and `b` reducers will receive `undefined` as *their* `state` arguments, and if they specify default `state` values, those will be returned.** This is how the combined `reducer` returns a `{ a: 'lol', b: 'wat' }` `state` object on the first invocation.

```
import { createStore } from 'redux'
const store = createStore(combined)
console.log(store.getState()) // { a: 'lol', b: 'wat' }
```

Let's consider a different scenario:

```
import { createStore } from 'redux'
const store = createStore(combined, { a: 'horse' })
console.log(store.getState()) // { a: 'horse', b: 'wat' }
```

Now I specified the `preloadedState` as the argument to `createStore()`. The `state` returned from the combined `reducer` combines the initial `state` I specified for the `a` `reducer` with the `'wat'` default argument specified that `b` `reducer` chose itself.

Let's recall what the combined `reducer` does:

```
// const combined = combineReducers({ a, b })
function combined(state = {}, action) {
  return {
    a: a(state.a, action),
    b: b(state.b, action)
  }
}
```

In this case, `state` was specified so it didn't fall back to `{}`. It was an object with `a` field equal to `'horse'`, but without the `b` field. This is why the `a` `reducer` received `'horse'` as its `state` and gladly returned it, but the `b` `reducer` received `undefined` as its `state` and thus returned *its idea* of the default `state` (in our example, `'wat'`). This is how we get `{ a: 'horse', b: 'wat' }` in return.

Recap

To sum this up, if you stick to Redux conventions and return the initial `state` from reducers when they're called with `undefined` as the `state` argument (the easiest way to implement this is to specify the `state` ES6 default argument value), you're going to have a nice useful behavior for combined reducers. **They will prefer the corresponding value in the `preloadedState` object you pass to the `createStore()` function, but if you didn't pass any, or if the corresponding field is not set, the default `state` argument specified by the `reducer` is chosen instead.** This approach works well because it provides both initialization and hydration of existing data, but lets individual reducers reset their `state` if their data was not preserved. Of course you can apply this pattern recursively, as you can use `combineReducers()` on many levels, or even compose reducers manually by calling reducers and giving them the relevant part of the `state` tree.

Using Immutable.JS with Redux

Table of Contents

- [Why should I use an immutable-focused library such as Immutable.JS?](#)
- [Why should I choose Immutable.JS as an immutable library?](#)
- [What are the issues with using Immutable.JS?](#)
- [Is Immutable.JS worth the effort?](#)
- [What are some opinionated Best Practices for using Immutable.JS with Redux?](#)

Why should I use an immutable-focused library such as Immutable.JS?

Immutable-focused libraries such as Immutable.JS have been designed to overcome the issues with immutability inherent within JavaScript, providing all the benefits of immutability with the performance your app requires.

Whether you choose to use such a library, or stick with plain JavaScript, depends on how comfortable you are with adding another dependency to your app, or how sure you are that you can avoid the pitfalls inherent within JavaScript's approach to immutability.

Whichever option you choose, make sure you're familiar with the concepts of [immutability](#), [side effects](#) and [mutation](#). In particular, ensure you have a deep understanding of what JavaScript does when updating and copying values in order to guard against accidental mutations that will degrade your app's performance, or break it altogether.

Further Information

Documentation

- [Recipes: immutability, side effects and mutation](#)

Articles

- [Introduction to Immutable.js and Functional Programming Concepts](#)
- [Pros and Cons of using immutability with React.js](#)

Why should I choose Immutable.JS as an immutable library?

Immutable.JS was designed to provide immutability in a performant manner in an effort to overcome the limitations of immutability with JavaScript. Its principle advantages include:

Guaranteed immutability

Data encapsulated in an Immutable.JS object is never mutated. A new copy is always returned. This contrasts with JavaScript, in which some operations do not mutate your data (e.g. some Array methods, including map, filter, concat, forEach, etc.), but some do (Array's pop, push, splice, etc.).

Rich API

Immutable.JS provides a rich set of immutable objects to encapsulate your data (e.g. Maps, Lists, Sets, Records, etc.), and an extensive set of methods to manipulate it, including methods to sort, filter, and group the data, reverse it, flatten it, and create subsets.

Performance

Immutable.JS does a lot of work behind the scenes to optimize performance. This is the key to its power, as using immutable data structures can involve a lot of expensive copying. In particular, immutably manipulating large, complex data sets, such as a nested Redux [state](#) tree, can generate many intermediate copies of objects, which consume memory and slow down performance as the browser's garbage collector fights to clean things up.

Immutable.JS avoids this by [cleverly sharing data structures](#) under the surface, minimizing the need to copy data. It also enables complex chains of operations to be carried out without creating unnecessary (and costly) cloned intermediate data that will quickly be thrown away.

You never see this, of course - the data you give to an Immutable.JS object is never mutated. Rather, it's the *intermediate* data generated within Immutable.JS from a chained sequence of method calls that is free to be mutated. You therefore get all the benefits of immutable data structures with none (or very little) of the potential performance hits.

Further Information

Articles

- [Immutable.js, persistent data structures and structural sharing](#)
- [PDF: JavaScript Immutability - Don't go changing](#)

Libraries

- [Immutable.js](#)

What are the issues with using Immutable.JS?

Although powerful, Immutable.JS needs to be used carefully, as it comes with issues of its own. Note, however, that all of these issues can be overcome quite easily with careful coding.

Difficult to interoperate with

JavaScript does not provide immutable data structures. As such, for Immutable.JS to provide its immutable guarantees, your data must be encapsulated within an Immutable.JS object (such as a `Map` or a `List`, etc.). Once it's contained in this way, it's hard for that data to then interoperate with other, plain JavaScript objects.

For example, you will no longer be able to reference an object's properties through standard JavaScript dot or bracket notation. Instead, you must reference them via Immutable.JS's `get()` or `getIn()` methods, which use an awkward syntax that accesses properties via an array of strings, each of which represents a property key.

For example, instead of `myObj.prop1.prop2.prop3`, you would use `myImmutableMap.getIn(['prop1', 'prop2', 'prop3'])`.

This makes it awkward to interoperate not just with your own code, but also with other libraries, such as `lodash` or `ramda`, that expect plain JavaScript objects.

Note that Immutable.JS objects do have a `toJS()` method, which returns the data as a plain JavaScript data structure, but this method is extremely slow, and using it extensively will negate the performance benefits that Immutable.JS provides

Once used, Immutable.JS will spread throughout your codebase

Once you encapsulate your data with Immutable.JS, you have to use Immutable.JS's `get()` or `getIn()` property accessors to access it.

This has the effect of spreading Immutable.JS across your entire codebase, including potentially your components, where you may prefer not to have such external dependencies. Your entire codebase must know what is, and what is not, an Immutable.JS object. It also makes removing Immutable.JS from your app difficult in the future, should you ever need to.

This issue can be avoided by [uncoupling your application logic from your data structures](#), as outlined in the [best practices section](#) below.

No Destructuring or Spread Operators

Because you must access your data via Immutable.JS's own `get()` and `getIn()` methods, you can no longer use JavaScript's destructuring operator (or the proposed Object spread operator), making your code more verbose.

Not suitable for small values that change often

Immutable.JS works best for collections of data, and the larger the better. It can be slow when your data comprises lots of small, simple JavaScript objects, with each comprising a few keys of primitive values.

Note, however, that this does not apply to the Redux [state](#) tree, which is (usually) represented as a large collection of data.

Difficult to Debug

Immutable.JS objects, such as `Map`, `List`, etc., can be difficult to debug, as inspecting such an object will reveal an entire nested hierarchy of Immutable.JS-specific properties that you don't care about, while your actual data that you do care about is encapsulated several layers deep.

To resolve this issue, use a browser extension such as the [Immutable.js Object Formatter](#), which surfaces your data in Chrome Dev Tools, and hides Immutable.JS's properties when inspecting your data.

Breaks object references, causing poor performance

One of the key advantages of immutability is that it enables shallow equality checking, which dramatically improves performance.

If two different variables reference the same immutable object, then a simple equality check of the two variables is enough to determine that they are equal, and that the object they both reference is unchanged. The equality check never has to check the values of any of the object's properties, as it is, of course, immutable.

However, shallow checking will not work if your data encapsulated within an Immutable.JS object is itself an object. This is because Immutable.JS's `toJS()` method, which returns the data contained within an Immutable.JS object as a JavaScript value, will create a new object every time it's called, and so break the reference with the

encapsulated data.

Accordingly, calling `toJS()` twice, for example, and assigning the result to two different variables will cause an equality check on those two variables to fail, even though the object values themselves haven't changed.

This is a particular issue if you use `toJS()` in a wrapped component's `mapStateToProps` function, as React-Redux shallowly compares each value in the returned props object. For example, the value referenced by the `todos` prop returned from `mapStateToProps` below will always be a different object, and so will fail a shallow equality check.

```
// AVOID .toJS() in mapStateToProps
function mapStateToProps(state) {
  return {
    todos: state.get('todos').toJS() // Always a new object
  }
}
```

When the shallow check fails, React-Redux will cause the component to re-render. Using `toJS()` in `mapStateToProps` in this way, therefore, will always cause the component to re-render, even if the value never changes, impacting heavily on performance.

This can be prevented by using `toJS()` in a Higher Order Component, as discussed in the [Best Practices section](#) below.

Further Information

Articles

- [Immutable.js, persistent data structures and structural sharing](#)
- [Immutable Data Structures and JavaScript](#)
- [React.js pure render performance anti-pattern](#)
- [Building Efficient UI with React and Redux](#)

Chrome Extension

- [Immutable Object Formatter](#)

Is Using Immutable.JS worth the effort?

Frequently, yes. There are various tradeoffs and opinions to consider, but there are many good reasons to use Immutable.JS. Do not underestimate the difficulty of trying to track down a property of your `state` tree that has been inadvertently mutated.

Components will both re-render when they shouldn't, and refuse to render when they should, and tracking down the bug causing the rendering issue is hard, as the component rendering incorrectly is not necessarily the one whose properties are being accidentally mutated.

This problem is caused predominantly by returning a mutated `state` object from a Redux `reducer`. With Immutable.JS, this problem simply does not exist, thereby removing a whole class of bugs from your app.

This, together with its performance and rich API for data manipulation, is why Immutable.JS is worth the effort.

Further Information

Documentation

- [Troubleshooting: Nothing happens when I dispatch an action](#)

What are some opinionated Best Practices for using Immutable.JS with Redux?

Immutable.JS can provide significant reliability and performance improvements to your app, but it must be used correctly. If you choose to use Immutable.JS (and remember, you are not required to, and there are other immutable libraries you can use), follow these opinionated best practices, and you'll be able to get the most out of it, without tripping up on any of the issues it can potentially cause.

Never mix plain JavaScript objects with Immutable.JS

Never let a plain JavaScript object contain Immutable.JS properties. Equally, never let an Immutable.JS object contain a plain JavaScript object.

Further Information

Articles

- [Immutable Data Structures and JavaScript](#)

Make your entire Redux state tree an Immutable.JS object

For a Redux app, your entire [state](#) tree should be an Immutable.JS object, with no plain JavaScript objects used at all.

- Create the tree using Immutable.JS's `fromJS()` function.
- Use an Immutable.JS-aware version of the `combineReducers` function, such as the one in [redux-immutable](#), as Redux itself expects the [state](#) tree to be a plain JavaScript object.
- When adding JavaScript objects to an Immutable.JS Map or List using Immutable.JS's `update`, `merge` or `set` methods, ensure that the object being added is first converted to an Immutable object using `fromJS()`.

Example

```
// avoid
const newObj = { key: value }
const newState = state.setIn(['prop1'], newObj)
// newObj has been added as a plain JavaScript object, NOT as an Immutable.JS Map

// recommended
const newObj = { key: value }
const newState = state.setIn(['prop1'], fromJS(newObj))
// newObj is now an Immutable.JS Map
```

Further Information

Articles

- [Immutable Data Structures and JavaScript](#)

Libraries

- [redux-immutable](#)

Use Immutable.JS everywhere except your dumb components

Using Immutable.JS everywhere keeps your code performant. Use it in your smart components, your selectors, your sagas or thunks, [action](#) creators, and especially your reducers.

Do not, however, use Immutable.JS in your dumb components.

Further Information

Articles

- [Immutable Data Structures and JavaScript](#)
- [Smart and Dumb Components in React](#)

Limit your use of `toJS()`

`toJS()` is an expensive function and negates the purpose of using Immutable.JS. Avoid its use.

Further Information

Discussions

- [Lee Byron on Twitter: "Perf tip for #immutablejs..."](#)

Your selectors should return Immutable.JS objects

Always. This practice has several advantages:

- It avoids unnecessary rerenders caused by calling `.toJS()` in selectors (since `.toJS()` will always return a new object).
 - It is possible to memoize selectors where you call `.toJS()`, but it's redundant when just returning Immutable.js objects without memoizing will suffice.
- It establishes a consistent interface for selectors; you won't have to keep track of whether an Immutable.js object or plain JavaScript object will be returned.

Use Immutable.JS objects in your Smart Components

Smart components that access the [store](#) via React Redux's `connect` function must use the Immutable.JS values returned by your selectors. Make sure you avoid the potential issues this can cause with unnecessary component re-rendering. Memoize your selectors using a library such as [reselect](#) if necessary.

Further Information

Documentation

- [Recipes: Computing Derived Data](#)
- [FAQ: Immutable Data](#)
- [Reselect Documentation: How do I use Reselect with Immutable.js?](#)

Articles

- [Redux Patterns and Anti-Patterns](#)

Libraries

- [Reselect: Selector library for Redux](#)

Never use `toJS()` in `mapStateToProps`

Converting an Immutable.JS object to a JavaScript object using `toJS()` will return a new object every time. If you do this in `mapStateToProps`, you will cause the component to believe that the object has changed every time the `state` tree changes, and so trigger an unnecessary re-render.

Further Information

Documentation

- [FAQ: Immutable Data](#)

Never use Immutable.JS in your Dumb Components

Your dumb components should be pure; that is, they should produce the same output given the same input, and have no external dependencies. If you pass such a component an Immutable.JS object as a prop, you make it dependent upon Immutable.JS to extract the prop's value and otherwise manipulate it.

Such a dependency renders the component impure, makes testing the component more difficult, and makes reusing and refactoring the component unnecessarily difficult.

Further Information

Articles

- [Immutable Data Structures and JavaScript](#)
- [Smart and Dumb Components in React](#)
- [Tips For a Better Redux Architecture: Lessons for Enterprise Scale](#)

Use a Higher Order Component to convert your Smart Component's Immutable.JS props to your Dumb Component's JavaScript props

Something needs to map the Immutable.JS props in your Smart Component to the pure JavaScript props used in your Dumb Component. That something is a Higher Order Component (HOC) that simply takes the Immutable.JS props from your Smart Component, and converts them using `toJS()` to plain JavaScript props, which are then passed to your Dumb Component.

An example of such a HOC follows. A similar HOC is available as an NPM package for your convenience: [with-immutable-props-to-js](#).

```
import React from 'react'
import { Iterable } from 'immutable'

export const toJS = WrappedComponent => wrappedComponentProps => {
  const KEY = 0
  const VALUE = 1
```

```

const propsJS = Object.entries(wrappedComponentProps).reduce(
  (newProps, wrappedComponentProp) => {
    newProps[wrappedComponentProp[KEY]] = Iterable.isIterable(
      wrappedComponentProp[VALUE]
    )
      ? wrappedComponentProp[VALUE].toJS()
      : wrappedComponentProp[VALUE]
    return newProps
  },
  {}
)

return <WrappedComponent {...propsJS} />
}

```

And this is how you would use it in your Smart Component:

```

import { connect } from 'react-redux'

import { toJS } from './to-js'
import DumbComponent from './dumb.component'

const mapStateToProps = state => {
  return {
    // obj is an Immutable object in Smart Component, but it's converted to a plain
    // JavaScript object by toJS, and so passed to DumbComponent as a pure JavaScript
    // object. Because it's still an Immutable.JS object here in mapStateToProps, though,
    // there is no issue with errant re-renderings.
    obj: getImmutableObjectFromStateTree(state)
  }
}

export default connect(mapStateToProps)(toJS(DumbComponent))

```

By converting Immutable.JS objects to plain JavaScript values within a HOC, we achieve Dumb Component portability, but without the performance hits of using `toJS()` in the Smart Component.

Note: if your app requires high performance, you may need to avoid `toJS()` altogether, and so will have to use Immutable.JS in your dumb components. However, for most apps this will not be the case, and the benefits of keeping Immutable.JS out of your dumb components (maintainability, portability and easier testing) will far outweigh any perceived performance improvements of keeping it in.

In addition, using `toJS` in a Higher Order Component should not cause much, if any, performance degradation, as the component will only be called when the connected component's props change. As with any performance issue, conduct performance checks first before deciding what to optimize.

Further Information

Documentation

- [React: Higher-Order Components](#)

Articles

- [React Higher Order Components in depth](#)

Discussions

- [Reddit: acemarke and cpsubrian comments on Dan Abramov: Redux is not an architecture or design pattern,](#)

[it is just a library.](#)

Gists

- [cpsubrian: React decorators for redux/react-router/immutable 'smart' components](#)

Use the Immutable Object Formatter Chrome Extension to Aid Debugging

Install the [Immutable Object Formatter](#) , and inspect your Immutable.JS data without seeing the noise of Immutable.JS's own object properties.

Further Information

Chrome Extension

- [Immutable Object Formatter](#)

Redux FAQ

Table of Contents

- **General**
 - [When should I learn Redux?](#)
 - [When should I use Redux?](#)
 - [Can Redux only be used with React?](#)
 - [Do I need to have a particular build tool to use Redux?](#)
- **Reducers**
 - [How do I share state between two reducers? Do I have to use combineReducers?](#)
 - [Do I have to use the switch statement to handle actions?](#)
- **Organizing State**
 - [Do I have to put all my state into Redux? Should I ever use React's setState\(\)?](#)
 - [Can I put functions, promises, or other non-serializable items in my store state?](#)
 - [How do I organize nested or duplicate data in my state?](#)
 - [Should I put form state or other UI state in my store?](#)
- **Store Setup**
 - [Can or should I create multiple stores? Can I import my store directly, and use it in components myself?](#)
 - [Is it OK to have more than one middleware chain in my store enhancer? What is the difference between next and dispatch in a middleware function?](#)
 - [How do I subscribe to only a portion of the state? Can I get the dispatched action as part of the subscription?](#)
- **Actions**
 - [Why should type be a string, or at least serializable? Why should my action types be constants?](#)
 - [Is there always a one-to-one mapping between reducers and actions?](#)
 - [How can I represent “side effects” such as AJAX calls? Why do we need things like “action creators”, “thunks”, and “middleware” to do async behavior?](#)
 - [What async middleware should I use? How do you decide between thunks, sagas, observables, or something else?](#)
 - [Should I dispatch multiple actions in a row from one action creator?](#)
- **Immutable Data**
 - [What are the benefits of immutability?](#)
 - [Why is immutability required by Redux?](#)
 - [What approaches are there for handling data immutability? Do I have to use Immutable.JS?](#)
 - [What are the issues with using JavaScript for immutable operations?](#)
- **Using Immutable.JS with Redux**
 - [Why should I use an immutable-focused library such as Immutable.JS?](#)
 - [Why should I choose Immutable.JS as an immutable library?](#)
 - [What are the issues with using Immutable.JS?](#)
 - [Is Immutable.JS worth the effort?](#)
 - [What are some opinionated Best Practices for using Immutable.JS with Redux?](#)
- **Code Structure**
 - [What should my file structure look like? How should I group my action creators and reducers in my project? Where should my selectors go?](#)
 - [How should I split my logic between reducers and action creators? Where should my “business logic”](#)

go?

- Why should I use action creators?
- Where should websockets and other persistent connections live?

- **Performance**

- How well does Redux “scale” in terms of performance and architecture?
- Won't calling “all my reducers” for each action be slow?
- Do I have to deep-clone my state in a reducer? Isn't copying my state going to be slow?
- How can I reduce the number of store update events?
- Will having “one state tree” cause memory problems? Will dispatching many actions take up memory?
- Will caching remote data cause memory problems?

- **Design Decisions**

- Why doesn't Redux pass the state and action to subscribers?
- Why doesn't Redux support using classes for actions and reducers?
- Why does the middleware signature use currying?
- Why does `applyMiddleware` use a closure for dispatch?
- Why doesn't `combineReducers` include a third argument with the entire state when it calls each reducer?
- Why doesn't `mapDispatchToProps` allow use of return values from `getState()` or `mapStateToProps()` ?

- **React Redux**

- Why should I use React-Redux?
- Why isn't my component re-rendering, or my `mapStateToProps` running?
- Why is my component re-rendering too often?
- How can I speed up my `mapStateToProps`?
- Why don't I have `this.props.dispatch` available in my connected component?
- Should I only connect my top component, or can I connect multiple components in my tree?

- **Miscellaneous**

- Are there any larger, “real” Redux projects?
- How can I implement authentication in Redux?

Redux FAQ: General

Table of Contents

- [When should I learn Redux?](#)
- [When should I use Redux?](#)
- [Can Redux only be used with React?](#)
- [Do I need to have a particular build tool to use Redux?](#)

General

When should I learn Redux?

What to learn can be an overwhelming question for a JavaScript developer. It helps to narrow the range of options by learning one thing at a time and focusing on problems you find in your work. Redux is a pattern for managing application [state](#). If you do not have problems with [state](#) management, you might find the benefits of Redux harder to understand. Some UI libraries (like React) have their own [state](#) management system. If you are using one of these libraries, especially if you are just learning to use them, we encourage you to learn the capabilities of that built-in system first. It might be all you need to build your application. If your application becomes so complex that you are confused about where [state](#) is stored or how [state](#) changes, then it is a good time to learn Redux. Experiencing the complexity that Redux seeks to abstract is the best preparation for effectively applying that abstraction to your work.

Further information

Articles

- [Deciding What Not To Learn](#)
- [How to learn web frameworks](#)
- [Redux vs MobX vs Flux vs... Do you even need that?](#)

Discussions

- [Ask HN: Overwhelmed with learning front-end, how do I proceed?](#)
- [Twitter: If you want to teach someone to use an abstraction...](#)
- [Twitter: it was never intended to be learned before...](#)
- [Twitter: Learning Redux before React?](#)
- [Twitter: The first time I used React, people told me I needed Redux...](#)
- [Twitter: This was my experience with Redux...](#)
- [Dev.to: When is it time to use Redux?](#)

When should I use Redux?

The need to use Redux should not be taken for granted.

As Pete Hunt, one of the early contributors to React, says:

You'll know when you need Flux. If you aren't sure if you need it, you don't need it.

Similarly, Dan Abramov, one of the creators of Redux, says:

I would like to amend this: don't use Redux until you have problems with vanilla React.

In general, use Redux when you have reasonable amounts of data changing over time, you need a single source of truth, and you find that approaches like keeping everything in a top-level React component's [state](#) are no longer sufficient.

However, it's also important to understand that using Redux comes with tradeoffs. It's not designed to be the shortest or fastest way to write code. It's intended to help answer the question "When did a certain slice of [state](#) change, and where did the data come from?", with predictable behavior. It does so by asking you to follow specific constraints in your application: [store](#) your application's [state](#) as plain data, describe changes as plain objects, and handle those changes with pure functions that apply updates immutably. This is often the source of complaints about "boilerplate". These constraints require effort on the part of a developer, but also open up a number of additional possibilities (such as [store](#) persistence and synchronization).

In the end, Redux is just a tool. It's a great tool, and there are some great reasons to use it, but there are also reasons you might not want to use it. Make informed decisions about your tools, and understand the tradeoffs involved in each decision.

Further information

Documentation

- [Introduction: Motivation](#)

Articles

- [React How-To](#)
- [You Might Not Need Redux](#)
- [The Case for Flux](#)
- [Some Reasons Why Redux is Useful in a React App](#)

Discussions

- [Twitter: Don't use Redux until...](#)
- [Twitter: Redux is designed to be predictable, not concise](#)
- [Twitter: Redux is useful to eliminate deep prop passing](#)
- [Twitter: Don't use Redux unless you're unhappy with local component state](#)
- [Twitter: You don't need Redux if your data never changes](#)
- [Twitter: If your reducer looks boring, don't use redux](#)
- [Reddit: You don't need Redux if your app just fetches something on a single page](#)
- [Stack Overflow: Why use Redux over Facebook Flux?](#)
- [Stack Overflow: Why should I use Redux in this example?](#)
- [Stack Overflow: What could be the downsides of using Redux instead of Flux?](#)
- [Stack Overflow: When should I add Redux to a React app?](#)
- [Stack Overflow: Redux vs plain React?](#)
- [Twitter: Redux is a platform for developers to build customized state management with reusable things](#)

Can Redux only be used with React?

Redux can be used as a data [store](#) for any UI layer. The most common usage is with React and React Native, but there are bindings available for Angular, Angular 2, Vue, Mithril, and more. Redux simply provides a subscription mechanism which can be used by any other code. That said, it is most useful when combined with a declarative view implementation that can infer the UI updates from the [state](#) changes, such as React or one of the similar libraries available.

Do I need to have a particular build tool to use Redux?

Redux is originally written in ES6 and transpiled for production into ES5 with Webpack and Babel. You should be able to use it regardless of your JavaScript build process. Redux also offers a UMD build that can be used directly without any build process at all. The [counter-vanilla](#) example demonstrates basic ES5 usage with Redux included as a `<script>` tag. As the relevant pull request says:

The new Counter Vanilla example is aimed to dispel the myth that Redux requires Webpack, React, hot reloading, sagas, [action](#) creators, constants, Babel, npm, CSS modules, decorators, fluent Latin, an Egghead subscription, a PhD, or an Exceeds Expectations O.W.L. level.

Nope, it's just HTML, some artisanal `<script>` tags, and plain old DOM manipulation. Enjoy!

Redux FAQ: Reducers

Table of Contents

- [How do I share state between two reducers? Do I have to use combineReducers?](#)
- [Do I have to use the switch statement to handle actions?](#)

Reducers

How do I share state between two reducers? Do I have to use combineReducers ?

The suggested structure for a Redux [store](#) is to split the [state](#) object into multiple “slices” or “domains” by key, and provide a separate [reducer](#) function to manage each individual data slice. This is similar to how the standard Flux pattern has multiple independent stores, and Redux provides the [combineReducers](#) utility function to make this pattern easier. However, it's important to note that [combineReducers](#) is *not* required—it is simply a utility function for the common use case of having a single [reducer](#) function per [state](#) slice, with plain JavaScript objects for the data.

Many users later want to try to share data between two reducers, but find that [combineReducers](#) does not allow them to do so. There are several approaches that can be used:

- If a [reducer](#) needs to know data from another slice of [state](#), the [state](#) tree shape may need to be reorganized so that a single [reducer](#) is handling more of the data.
- You may need to write some custom functions for handling some of these actions. This may require replacing [combineReducers](#) with your own top-level [reducer](#) function. You can also use a utility such as [reduce-reducers](#) to run [combineReducers](#) to handle most actions, but also run a more specialized [reducer](#) for specific actions that cross [state](#) slices.
- [Async action creators](#) such as [redux-thunk](#) have access to the entire [state](#) through [getState\(\)](#) . An [action](#) creator can retrieve additional data from the [state](#) and put it in an [action](#), so that each [reducer](#) has enough information to update its own [state](#) slice.

In general, remember that reducers are just functions—you can organize them and subdivide them any way you want, and you are encouraged to break them down into smaller, reusable functions (“[reducer](#) composition”). While you do so, you may pass a custom third argument from a parent [reducer](#) if a child [reducer](#) needs additional data to calculate its next [state](#). You just need to make sure that together they follow the basic rules of reducers: `(state, action) => newState` , and update [state](#) immutably rather than mutating it directly.

Further information

Documentation

- [API: combineReducers](#)
- [Recipes: Structuring Reducers](#)

Discussions

- [#601: A concern on combineReducers, when an action is related to multiple reducers](#)

- [#1400: Is passing top-level state object to branch reducer an anti-pattern?](#)
- [Stack Overflow: Accessing other parts of the state when using combined reducers?](#)
- [Stack Overflow: Reducing an entire subtree with redux combineReducers](#)
- [Sharing State Between Redux Reducers](#)

Do I have to use the `switch` statement to handle actions?

No. You are welcome to use any approach you'd like to respond to an [action](#) in a [reducer](#). The `switch` statement is the most common approach, but it's fine to use `if` statements, a lookup table of functions, or to create a function that abstracts this away. In fact, while Redux does require that [action](#) objects contain a `type` field, your [reducer](#) logic doesn't even have to rely on that to handle the [action](#). That said, the standard approach is definitely using a switch statement or a lookup table based on `type`.

Further information

Documentation

- [Recipes: Reducing Boilerplate](#)
- [Recipes: Structuring Reducers - Splitting Reducer Logic](#)

Discussions

- [#883: take away the huge switch block](#)
- [#1167: Reducer without switch](#)

Redux FAQ: Organizing State

Table of Contents

- [Do I have to put all my state into Redux? Should I ever use React's setState\(\)?](#)
- [Can I put functions, promises, or other non-serializable items in my store state?](#)
- [How do I organize nested or duplicate data in my state?](#)
- [Should I put form state or other UI state in my store?](#)

Organizing State

Do I have to put all my state into Redux? Should I ever use React's setState() ?

There is no “right” answer for this. Some users prefer to keep every single piece of data in Redux, to maintain a fully serializable and controlled version of their application at all times. Others prefer to keep non-critical or UI [state](#), such as “is this dropdown currently open”, inside a component's internal [state](#).

Using local component [state](#) is fine. As a developer, it is *your* job to determine what kinds of [state](#) make up your application, and where each piece of [state](#) should live. Find a balance that works for you, and go with it.

Some common rules of thumb for determining what kind of data should be put into Redux:

- Do other parts of the application care about this data?
- Do you need to be able to create further derived data based on this original data?
- Is the same data being used to drive multiple components?
- Is there value to you in being able to restore this [state](#) to a given point in time (ie, time travel debugging)?
- Do you want to cache the data (ie, use what's in [state](#) if it's already there instead of re-requesting it)?
- Do you want to keep this data consistent while hot-reloading UI components (which may lose their internal [state](#) when swapped)?

There are a number of community packages that implement various approaches for storing per-component [state](#) in a Redux [store](#) instead, such as [redux-ui](#), [redux-component](#), [redux-react-local](#), and more. It's also possible to apply Redux's principles and concept of reducers to the task of updating local component [state](#) as well, along the lines of `this.setState((previousState) => reducer(previousState, someAction))`.

Further information

Articles

- [You Might Not Need Redux](#)
- [Finding \[state\]\(#\) 's place with React and Redux](#)
- [A Case for setState](#)
- [How to handle state in React: the missing FAQ](#)
- [Where to Hold React Component Data: state, store, static, and this](#)
- [The 5 Types of React Application State](#)
- [Shape Your Redux Store Like Your Database](#)

Discussions

- [#159: Investigate using Redux for pseudo-local component state](#)
- [#1098: Using Redux in reusable React component](#)
- [#1287: How to choose between Redux's store and React's state?](#)
- [#1385: What are the disadvantages of storing all your state in a single immutable atom?](#)
- [Twitter: Should I keep something in React component state?](#)
- [Twitter: Using a reducer to update a component](#)
- [React Forums: Redux and global state vs local state](#)
- [Reddit: "When should I put something into my Redux store?"](#)
- [Stack Overflow: Why is state all in one place, even state that isn't global?](#)
- [Stack Overflow: Should all component state be kept in Redux store?](#)

Libraries

- [Redux Addons Catalog: Component State](#)

Can I put functions, promises, or other non-serializable items in my store state?

It is highly recommended that you only put plain serializable objects, arrays, and primitives into your [store](#). It's *technically* possible to insert non-serializable items into the [store](#), but doing so can break the ability to persist and rehydrate the contents of a [store](#), as well as interfere with time-travel debugging.

If you are okay with things like persistence and time-travel debugging potentially not working as intended, then you are totally welcome to put non-serializable items into your Redux [store](#). Ultimately, it's *your* application, and how you implement it is up to you. As with many other things about Redux, just be sure you understand what tradeoffs are involved.

Further information

Discussions

- [#1248: Is it ok and possible to store a react component in a reducer?](#)
- [#1279: Have any suggestions for where to put a Map Component in Flux?](#)
- [#1390: Component Loading](#)
- [#1407: Just sharing a great base class](#)
- [#1793: React Elements in Redux State](#)

How do I organize nested or duplicate data in my state?

Data with IDs, nesting, or relationships should generally be stored in a “normalized” fashion: each object should be stored once, keyed by ID, and other objects that reference it should only [store](#) the ID rather than a copy of the entire object. It may help to think of parts of your [store](#) as a database, with individual “tables” per item type. Libraries such as [normalizr](#) and [redux-orm](#) can provide help and abstractions in managing normalized data.

Further information

Documentation

- [Advanced: Async Actions](#)
- [Examples: Real World example](#)

- [Recipes: Structuring Reducers - Prerequisite Concepts](#)
- [Recipes: Structuring Reducers - Normalizing State Shape](#)
- [Examples: Tree View](#)

Articles

- [High-Performance Redux](#)
- [Querying a Redux Store](#)

Discussions

- [#316: How to create nested reducers?](#)
- [#815: Working with Data Structures](#)
- [#946: Best way to update related state fields with split reducers?](#)
- [#994: How to cut the boilerplate when updating nested entities?](#)
- [#1255: Normalizr usage with nested objects in React/Redux](#)
- [#1269: Add tree view example](#)
- [#1824: Normalising state and garbage collection](#)
- [Twitter: state shape should be normalized](#)
- [Stack Overflow: How to handle tree-shaped entities in Redux reducers?](#)
- [Stack Overflow: How to optimize small updates to props of nested components in React + Redux?](#)

Should I put form state or other UI state in my store?

The [same rules of thumb for deciding what should go in the Redux store](#) apply for this question as well.

Based on those rules of thumb, most form [state](#) doesn't need to go into Redux, as it's probably not being shared between components. However, that decision is always going to be specific to you and your application. You might choose to keep some form [state](#) in Redux because you are editing data that came from the [store](#) originally, or because you do need to see the work-in-progress values reflected in other components elsewhere in the application. On the other hand, it may be a lot simpler to keep the form [state](#) local to the component, and only dispatch an [action](#) to put the data in the [store](#) once the user is done with the form.

Based on this, in most cases you probably don't need a Redux-based form management library either. We suggest trying these approaches, in this order:

- Even if the data is coming from the Redux [store](#), start by writing your form logic by hand. It's likely this is all you'll need. (See [Gosha Arinich's posts on working with forms in React](#) for some excellent guidance on this.)
- If you decide that writing forms "manually" is too difficult, try a React-based form library like [Formik](#) or [React-Final-Form](#).
- If you are absolutely sure you *must* use a Redux-based form library because the other approaches aren't sufficient, then you may finally want to look at [Redux-Form](#) and [React-Redux-Form](#).

If you are keeping form [state](#) in Redux, you should take some time to consider performance characteristics. Dispatching an [action](#) on every keystroke of a text input probably isn't worthwhile, and you may want to look into [ways to buffer keystrokes to keep changes local before dispatching](#). As always, take some time to analyze the overall performance needs of your own application.

Other kinds of UI [state](#) follow these rules of thumb as well. The classic example is tracking an `isDropdownOpen` flag. In most situations, the rest of the app doesn't care about this, so in most cases it should stay in component [state](#). However, depending on your application, it may make sense to use Redux to [manage dialogs and other popups](#), tabs, expanding panels, and so on.

Further Information

Articles

- [Gosha Arinich: Writings on Forms in React](#)
- [Practical Redux, Part 6: Connected Lists and Forms](#)
- [Practical Redux, Part 7: Form Change Handling](#)
- [Practical Redux, Part 10: Managing Modals and Context Menus](#)
- [React/Redux Links: Redux UI Management](#)

Redux FAQ: Store Setup

Table of Contents

- [Can or should I create multiple stores? Can I import my store directly, and use it in components myself?](#)
- [Is it OK to have more than one middleware chain in my store enhancer? What is the difference between next and dispatch in a middleware function?](#)
- [How do I subscribe to only a portion of the state? Can I get the dispatched action as part of the subscription?](#)

Store Setup

Can or should I create multiple stores? Can I import my store directly, and use it in components myself?

The original Flux pattern describes having multiple “stores” in an app, each one holding a different area of domain data. This can introduce issues such as needing to have one `store` “`waitFor`” another `store` to update. This is not necessary in Redux because the separation between data domains is already achieved by splitting a single `reducer` into smaller reducers.

As with several other questions, it is *possible* to create multiple distinct Redux stores in a page, but the intended pattern is to have only a single `store`. Having a single `store` enables using the Redux DevTools, makes persisting and rehydrating data simpler, and simplifies the subscription logic.

Some valid reasons for using multiple stores in Redux might include:

- Solving a performance issue caused by too frequent updates of some part of the `state`, when confirmed by profiling the app.
- Isolating a Redux app as a component in a bigger application, in which case you might want to create a `store` per root component instance.

However, creating new stores shouldn't be your first instinct, especially if you come from a Flux background. Try `reducer` composition first, and only use multiple stores if it doesn't solve your problem.

Similarly, while you *can* reference your `store` instance by importing it directly, this is not a recommended pattern in Redux. If you create a `store` instance and export it from a module, it will become a singleton. This means it will be harder to isolate a Redux app as a component of a larger app, if this is ever necessary, or to enable server rendering, because on the server you want to create separate `store` instances for every request.

With [React Redux](#), the wrapper classes generated by the `connect()` function do actually look for `props.store` if it exists, but it's best if you wrap your root component in `<Provider store={store}>` and let React Redux worry about passing the `store` down. This way components don't need to worry about importing a `store` module, and isolating a Redux app or enabling server rendering is much easier to do later.

Further information

Documentation

- [API: Store](#)

Discussions

- [#1346: Is it bad practice to just have a 'stores' directory?](#)
- [Stack Overflow: Redux multiple stores, why not?](#)
- [Stack Overflow: Accessing Redux state in an action creator](#)
- [Gist: Breaking out of Redux paradigm to isolate apps](#)

Is it OK to have more than one middleware chain in my store enhancer? What is the difference between `next` and `dispatch` in a middleware function?

Redux [middleware](#) act like a linked list. Each [middleware](#) function can either call `next(action)` to pass an [action](#) along to the next [middleware](#) in line, call `dispatch(action)` to restart the processing at the beginning of the list, or do nothing at all to stop the [action](#) from being processed further.

This chain of [middleware](#) is defined by the arguments passed to the `applyMiddleware` function used when creating a [store](#). Defining multiple chains will not work correctly, as they would have distinctly different `dispatch` references and the different chains would effectively be disconnected.

Further information

Documentation

- [Advanced: Middleware](#)
- [API: applyMiddleware](#)

Discussions

- [#1051: Shortcomings of the current applyMiddleware and composing createStore](#)
- [Understanding Redux Middleware](#)
- [Exploring Redux Middleware](#)

How do I subscribe to only a portion of the state? Can I get the dispatched action as part of the subscription?

Redux provides a single `store.subscribe` method for notifying listeners that the [store](#) has updated. Listener callbacks do not receive the current [state](#) as an argument—it is simply an indication that *something* has changed. The subscriber logic can then call `getState()` to get the current [state](#) value.

This API is intended as a low-level primitive with no dependencies or complications, and can be used to build higher-level subscription logic. UI bindings such as React Redux can create a subscription for each connected component. It is also possible to write functions that can intelligently compare the old [state](#) vs the new [state](#), and execute additional logic if certain pieces have changed. Examples include [redux-watch](#), [redux-subscribe](#) and [redux-subscriber](#) which offer different approaches to specifying subscriptions and handling changes.

The new [state](#) is not passed to the listeners in order to simplify implementing [store](#) enhancers such as the Redux DevTools. In addition, subscribers are intended to react to the [state](#) value itself, not the [action](#). [Middleware](#) can be used if the [action](#) is important and needs to be handled specifically.

Further information

Documentation

- [Basics: Store](#)
- [API: Store](#)

Discussions

- [#303: subscribe API with state as an argument](#)
- [#580: Is it possible to get action and state in store.subscribe?](#)
- [#922: Proposal: add subscribe to middleware API](#)
- [#1057: subscribe listener can get action param?](#)
- [#1300: Redux is great but major feature is missing](#)

Libraries

- [Redux Addons Catalog: Store Change Subscriptions](#)

Redux FAQ: Actions

Table of Contents

- [Why should type be a string, or at least serializable? Why should my action types be constants?](#)
- [Is there always a one-to-one mapping between reducers and actions?](#)
- [How can I represent “side effects” such as AJAX calls? Why do we need things like “action creators”, “thunks”, and “middleware” to do async behavior?](#)
- [What async middleware should I use? How do you decide between thunks, sagas, observables, or something else?](#)
- [Should I dispatch multiple actions in a row from one action creator?](#)

Actions

Why should `type` be a string, or at least serializable? Why should my action types be constants?

As with [state](#), serializable actions enable several of Redux's defining features, such as time travel debugging, and recording and replaying actions. Using something like a `Symbol` for the `type` value or using `instanceof` checks for actions themselves would break that. Strings are serializable and easily self-descriptive, and so are a better choice. Note that it is okay to use Symbols, Promises, or other non-serializable values in an [action](#) if the [action](#) is intended for use by [middleware](#). Actions only need to be serializable by the time they actually reach the [store](#) and are passed to the reducers.

We can't reliably enforce serializable actions for performance reasons, so Redux only checks that every [action](#) is a plain object, and that the `type` is defined. The rest is up to you, but you might find that keeping everything serializable helps debug and reproduce issues.

Encapsulating and centralizing commonly used pieces of code is a key concept in programming. While it is certainly possible to manually create [action](#) objects everywhere, and write each `type` value by hand, defining reusable constants makes maintaining code easier. If you put constants in a separate file, you can [check your `import` statements against typos](#) so you can't accidentally use the wrong string.

Further information

Documentation

- [Reducing Boilerplate](#)

Discussion

- [#384: Recommend that Action constants be named in the past tense](#)
- [#628: Solution for simple action creation with less boilerplate](#)
- [#1024: Proposal: Declarative reducers](#)
- [#1167: Reducer without switch](#)
- [Stack Overflow: Why do you need 'Actions' as data in Redux?](#)
- [Stack Overflow: What is the point of the constants in Redux?](#)

Is there always a one-to-one mapping between reducers and actions?

No. We suggest you write independent small [reducer](#) functions that are each responsible for updates to a specific slice of [state](#). We call this pattern “[reducer](#) composition”. A given [action](#) could be handled by all, some, or none of them. This keeps components decoupled from the actual data changes, as one [action](#) may affect different parts of the [state](#) tree, and there is no need for the component to be aware of this. Some users do choose to bind them more tightly together, such as the “ducks” file structure, but there is definitely no one-to-one mapping by default, and you should break out of such a paradigm any time you feel you want to handle an [action](#) in many reducers.

Further information

Documentation

- [Basics: Reducers](#)
- [Recipes: Structuring Reducers](#)

Discussions

- [Twitter: most common Redux misconception](#)
- [#1167: Reducer without switch](#)
- [Reduxible #8: Reducers and action creators aren't a one-to-one mapping](#)
- [Stack Overflow: Can I dispatch multiple actions without Redux Thunk middleware?](#)

How can I represent “side effects” such as AJAX calls? Why do we need things like “action creators”, “thunks”, and “middleware” to do async behavior?

This is a long and complex topic, with a wide variety of opinions on how code should be organized and what approaches should be used.

Any meaningful web app needs to execute complex logic, usually including asynchronous work such as making AJAX requests. That code is no longer purely a function of its inputs, and the interactions with the outside world are known as “[side effects](#)”

Redux is inspired by functional programming, and out of the box, has no place for side effects to be executed. In particular, [reducer](#) functions *must* always be pure functions of `(state, action) => newState`. However, Redux's [middleware](#) makes it possible to intercept dispatched actions and add additional complex behavior around them, including side effects.

In general, Redux suggests that code with side effects should be part of the [action](#) creation process. While that logic *can* be performed inside of a UI component, it generally makes sense to extract that logic into a reusable function so that the same logic can be called from multiple places—in other words, an [action](#) creator function.

The simplest and most common way to do this is to add the [Redux Thunk middleware](#) that lets you write [action](#) creators with more complex and asynchronous logic. Another widely-used method is [Redux Saga](#) which lets you write more synchronous-looking code using generators, and can act like “background threads” or “daemons” in a Redux app. Yet another approach is [Redux Loop](#), which inverts the process by allowing your reducers to declare side effects in response to [state](#) changes and have them executed separately. Beyond that, there are *many* other community-developed libraries and ideas, each with their own take on how side effects should be managed.

Further information

Documentation

- [Advanced: Async Actions](#)
- [Advanced: Async Flow](#)
- [Advanced: Middleware](#)

Articles

- [Redux Side-Effects and You](#)
- [Pure functionality and side effects in Redux](#)
- [From Flux to Redux: Async Actions the easy way](#)
- [React/Redux Links: "Redux Side Effects" category](#)
- [Gist: Redux-Thunk examples](#)

Discussions

- [#291: Trying to put API calls in the right place](#)
- [#455: Modeling side effects](#)
- [#533: Simpler introduction to async action creators](#)
- [#569: Proposal: API for explicit side effects](#)
- [#1139: An alternative side effect model based on generators and sagas](#)
- [Stack Overflow: Why do we need middleware for async flow in Redux?](#)
- [Stack Overflow: How to dispatch a Redux action with a timeout?](#)
- [Stack Overflow: Where should I put synchronous side effects linked to actions in redux?](#)
- [Stack Overflow: How to handle complex side-effects in Redux?](#)
- [Stack Overflow: How to unit test async Redux actions to mock ajax response](#)
- [Stack Overflow: How to fire AJAX calls in response to the state changes with Redux?](#)
- [Reddit: Help performing Async API calls with Redux-Promise Middleware.](#)
- [Twitter: possible comparison between sagas, loops, and other approaches](#)

What async middleware should I use? How do you decide between thunks, sagas, observables, or something else?

There are [many async/side effect middlewares available](#), but the most commonly used ones are [redux-thunk](#), [redux-saga](#), and [redux-observable](#). These are different tools, with different strengths, weaknesses, and use cases.

As a general rule of thumb:

- Thunks are best for complex synchronous logic (especially code that needs access to the entire Redux [store state](#)), and simple async logic (like basic AJAX calls). With the use of `async/await`, it can be reasonable to use thunks for some more complex promise-based logic as well.
- Sagas are best for complex async logic and decoupled "background thread"-type behavior, especially if you need to listen to dispatched actions (which is something that can't be done with thunks). They require familiarity with ES6 generator functions and `redux-saga`'s "effects" operators.
- Observables solve the same problems as sagas, but rely on RxJS to implement async behavior. They require familiarity with the RxJS API.

We recommend that most Redux users should start with thunks, and then add an additional side effect library like sagas or observables later if their app really requires handling for more complex async logic.

Since sagas and observables have the same use case, an application would normally use one or the other, but not both. However, note that **it's absolutely fine to use both thunks and either sagas or observables together**, because they solve different problems.

Articles

- [Decembersoft: What is the right way to do asynchronous operations in Redux?](#)
- [Decembersoft: Redux-Thunk vs Redux-Saga](#)
- [Redux-Thunk vs Redux-Saga: an overview](#)
- [Redux-Saga V.S. Redux-Observable](#)

Discussions

- [Reddit: discussion of using thunks and sagas together, and pros and cons of sagas](#)
- [Stack Overflow: Pros/cons of using redux-saga with ES6 generators vs redux-thunk with ES2017 async/await](#)
- [Stack Overflow: Why use Redux-Observable over Redux-Saga?](#)

Should I dispatch multiple actions in a row from one action creator?

There's no specific rule for how you should structure your actions. Using an async [middleware](#) like Redux Thunk certainly enables scenarios such as dispatching multiple distinct but related actions in a row, dispatching actions to represent progression of an AJAX request, dispatching actions conditionally based on [state](#), or even dispatching an [action](#) and checking the updated [state](#) immediately afterwards.

In general, ask if these actions are related but independent, or should actually be represented as one [action](#). Do what makes sense for your own situation but try to balance the readability of reducers with readability of the [action](#) log. For example, an [action](#) that includes the whole new [state](#) tree would make your [reducer](#) a one-liner, but the downside is now you have no history of *why* the changes are happening, so debugging gets really difficult. On the other hand, if you emit actions in a loop to keep them granular, it's a sign that you might want to introduce a new [action](#) type that is handled in a different way.

Try to avoid dispatching several times synchronously in a row in the places where you're concerned about performance. There are a number of addons and approaches that can batch up dispatches as well.

Further information

Documentation

- [FAQ: Performance - Reducing Update Events](#)

Articles

- [Idiomatic Redux: Thoughts on Thunks, Sagas, Abstraction, and Reusability](#)

Discussions

- [#597: Valid to dispatch multiple actions from an event handler?](#)
- [#959: Multiple actions one dispatch?](#)
- [Stack Overflow: Should I use one or several action types to represent this async action?](#)
- [Stack Overflow: Do events and actions have a 1:1 relationship in Redux?](#)
- [Stack Overflow: Should actions be handled by reducers to related actions or generated by action creators themselves?](#)
- [Twitter: "Good thread on the problems with Redux Thunk..."](#)

Redux FAQ: Immutable Data

Table of Contents

- [What are the benefits of immutability?](#)
- [Why is immutability required by Redux?](#)
- [Why does Redux's use of shallow equality checking require immutability?](#)
 - [How do Shallow and Deep Equality Checking differ?](#)
 - [How does Redux use shallow equality checking?](#)
 - [How does `combineReducers` use shallow equality checking?](#)
 - [How does React-Redux use shallow equality checking?](#)
 - [How does React-Redux use shallow equality checking to determine whether a component needs re-rendering?](#)
 - [Why will shallow equality checking not work with mutable objects?](#)
 - [Does shallow equality checking with a mutable object cause problems with Redux?](#)
 - [Why does a reducer mutating the state prevent React-Redux from re-rendering a wrapped component?](#)
 - [Why does a selector mutating and returning a persistent object to `mapStateToProps` prevent React-Redux from re-rendering a wrapped component?](#)
 - [How does immutability enable a shallow check to detect object mutations?](#)
 - [How can immutability in your reducers cause components to render unnecessarily?](#)
 - [How can immutability in `mapStateToProps` cause components to render unnecessarily?](#)
- [What approaches are there for handling data immutability? Do I have to use Immutable.js?](#)
- [What are the issues with using JavaScript for immutable operations?](#)

What are the benefits of immutability?

Immutability can bring increased performance to your app, and leads to simpler programming and debugging, as data that never changes is easier to reason about than data that is free to be changed arbitrarily throughout your app.

In particular, immutability in the context of a Web app enables sophisticated change detection techniques to be implemented simply and cheaply, ensuring the computationally expensive process of updating the DOM occurs only when it absolutely has to (a cornerstone of React's performance improvements over other libraries).

Further information

Articles

- [Introduction to Immutable.js and Functional Programming Concepts](#)
- [JavaScript Immutability presentation \(PDF - see slide 12 for benefits\)](#)
- [Immutable.js - Immutable Collections for JavaScript](#)
- [React: Optimizing Performance](#)
- [JavaScript Application Architecture On The Road To 2015](#)

Why is immutability required by Redux?

- Both Redux and React-Redux employ [shallow equality checking](#). In particular: - Redux's `combineReducers` utility [shallowly checks for reference changes](#) caused by the reducers that it calls. - React-Redux's `connect` method generates components that [shallowly check reference changes to the root state](#), and the return values from the `mapStateToProps` function to see if the wrapped components actually need to re-render. Such [shallow checking requires immutability](#) to function correctly.
- Immutable data management ultimately makes data handling safer.
- Time-travel debugging requires that reducers be pure functions with no side effects, so that you can correctly jump between different states.

Further Information

Documentation

- [Recipes: Prerequisite Reducer Concepts](#)

Discussions

- [Reddit: Why Redux Needs Reducers To Be Pure Functions](#)

Why does Redux's use of shallow equality checking require immutability?

Redux's use of shallow equality checking requires immutability if any connected components are to be updated correctly. To see why, we need to understand the difference between shallow and deep equality checking in JavaScript.

How do shallow and deep equality checking differ?

Shallow equality checking (or *reference equality*) simply checks that two different *variables* reference the same object; in contrast, deep equality checking (or *value equality*) must check every *value* of two objects' properties.

A shallow equality check is therefore as simple (and as fast) as `a === b`, whereas a deep equality check involves a recursive traversal through the properties of two objects, comparing the value of each property at each step.

It's for this improvement in performance that Redux uses shallow equality checking.

Further Information

Articles

- [Pros and Cons of using immutability with React.js](#)

How does Redux use shallow equality checking?

Redux uses shallow equality checking in its `combineReducers` function to return either a new mutated copy of the root `state` object, or, if no mutations have been made, the current root `state` object.

Further Information

Documentation

- [API: combineReducers](#)

How does `combineReducers` use shallow equality checking?

The [suggested structure](#) for a Redux [store](#) is to split the [state](#) object into multiple "slices" or "domains" by key, and provide a separate [reducer](#) function to manage each individual data slice.

`combineReducers` makes working with this style of structure easier by taking a `reducers` argument that's defined as a hash table comprising a set of key/value pairs, where each key is the name of a [state](#) slice, and the corresponding value is the [reducer](#) function that will act on it.

So, for example, if your [state](#) shape is `{ todos, counter }`, the call to `combineReducers` would be:

```
combineReducers({ todos: myTodosReducer, counter: myCounterReducer })
```

where:

- the keys `todos` and `counter` each refer to a separate [state](#) slice;
- the values `myTodosReducer` and `myCounterReducer` are [reducer](#) functions, with each acting on the [state](#) slice identified by the respective key.

`combineReducers` iterates through each of these key/value pairs. For each iteration, it:

- creates a reference to the current [state](#) slice referred to by each key;
- calls the appropriate [reducer](#) and passes it the slice;
- creates a reference to the possibly-mutated [state](#) slice that's returned by the [reducer](#).

As it continues through the iterations, `combineReducers` will construct a new [state](#) object with the [state](#) slices returned from each [reducer](#). This new [state](#) object may or may not be different from the current [state](#) object. It is here that `combineReducers` uses shallow equality checking to determine whether the [state](#) has changed.

Specifically, at each stage of the iteration, `combineReducers` performs a shallow equality check on the current [state](#) slice and the [state](#) slice returned from the [reducer](#). If the [reducer](#) returns a new object, the shallow equality check will fail, and `combineReducers` will set a `hasChanged` flag to true.

After the iterations have completed, `combineReducers` will check the [state](#) of the `hasChanged` flag. If it's true, the newly-constructed [state](#) object will be returned. If it's false, the *current state* object is returned.

This is worth emphasizing: *If the reducers all return the same [state](#) object passed to them, then `combineReducers` will return the current root [state](#) object, not the newly updated one.*

Further Information

Documentation

- [API: combineReducers](#)
- [Redux FAQ - How do I share state between two reducers? do I have to use `combineReducers` ?](#)

Video

- [Egghead.io: Redux: Implementing combineReducers\(\) from Scratch](#)

How does React-Redux use shallow equality checking?

React-Redux uses shallow equality checking to determine whether the component it's wrapping needs to be re-rendered.

To do this, it assumes that the wrapped component is pure; that is, that the component will produce the [same results given the same props and state](#).

By assuming the wrapped component is pure, it need only check whether the root [state](#) object or the values returned from `mapStateToProps` have changed. If they haven't, the wrapped component does not need re-rendering.

It detects a change by keeping a reference to the root [state](#) object, and a reference to *each value* in the props object that's returned from the `mapStateToProps` function.

It then runs a shallow equality check on its reference to the root [state](#) object and the [state](#) object passed to it, and a separate series of shallow checks on each reference to the props object's values and those that are returned from running the `mapStateToProps` function again.

Further Information

Documentation

- [React-Redux Bindings](#)

Articles

- [API: React-Redux's connect function and mapStateToProps](#)
- [Redux FAQ: Why isn't my component re-rendering, or my mapStateToProps running?](#)

Why does React-Redux shallowly check each value within the props object returned from mapStateToProps ?

React-Redux performs a shallow equality check on each *value* within the props object, not on the props object itself.

It does so because the props object is actually a hash of prop names and their values (or selector functions that are used to retrieve or generate the values), such as in this example:

```
function mapStateToProps(state) {
  return {
    todos: state.todos, // prop value
    visibleTodos: getVisibleTodos(state) // selector
  }
}

export default connect(mapStateToProps)(TodoApp)
```

As such, a shallow equality check of the props object returned from repeated calls to `mapStateToProps` would always fail, as a new object would be returned each time.

React-Redux therefore maintains separate references to each *value* in the returned props object.

Further Information

Articles

- [React.js pure render performance anti-pattern](#)

How does React-Redux use shallow equality checking to determine whether a component needs re-rendering?

Each time React-Redux's `connect` function is called, it will perform a shallow equality check on its stored reference to the root `state` object, and the current root `state` object passed to it from the `store`. If the check passes, the root `state` object has not been updated, and so there is no need to re-render the component, or even call `mapStateToProps`.

If the check fails, however, the root `state` object *has* been updated, and so `connect` will call `mapStateToProps` to see if the props for the wrapped component have been updated.

It does this by performing a shallow equality check on each value within the object individually, and will only trigger a re-render if one of those checks fails.

In the example below, if `state.todos` and the value returned from `getVisibleTodos()` do not change on successive calls to `connect`, then the component will not re-render.

```
function mapStateToProps(state) {
  return {
    todos: state.todos, // prop value
    visibleTodos: getVisibleTodos(state) // selector
  }
}

export default connect(mapStateToProps)(TodoApp)
```

Conversely, in this next example (below), the component will *always* re-render, as the value of `todos` is always a new object, regardless of whether or not its values change:

```
// AVOID - will always cause a re-render
function mapStateToProps(state) {
  return {
    // todos always references a newly-created object
    todos: {
      all: state.todos,
      visibleTodos: getVisibleTodos(state)
    }
  }
}

export default connect(mapStateToProps)(TodoApp)
```

If the shallow equality check fails between the new values returned from `mapStateToProps` and the previous values that React-Redux kept a reference to, then a re-rendering of the component will be triggered.

Further Information

Articles

- [Practical Redux, Part 6: Connected Lists, Forms, and Performance](#)
- [React.js Pure Render Performance Anti-Pattern](#)
- [High Performance Redux Apps](#)

Discussions

- [#1816: Component connected to state with `mapStateToProps`](#)

- [#300: Potential connect\(\) optimization](#)

Why will shallow equality checking not work with mutable objects?

Shallow equality checking cannot be used to detect if a function mutates an object passed into it if that object is mutable.

This is because two variables that reference the same object will *always* be equal, regardless of whether the object's values changes or not, as they're both referencing the same object. Thus, the following will always return true:

```
function mutateObj(obj) {  
  obj.key = 'newValue'  
  return obj  
}  
  
const param = { key: 'originalValue' }  
const returnVal = mutateObj(param)  
  
param === returnVal  
//> true
```

The shallow check of `param` and `returnVal` simply checks whether both variables reference the same object, which they do. `mutateObj()` may return a mutated version of `obj`, but it's still the same object as that passed in. The fact that its values have been changed within `mutateObj` matters not at all to a shallow check.

Further Information

Articles

- [Pros and Cons of using immutability with React.js](#)

Does shallow equality checking with a mutable object cause problems with Redux?

Shallow equality checking with a mutable object will not cause problems with Redux, but [it will cause problems with libraries that depend on the store, such as React-Redux](#).

Specifically, if the `state` slice passed to a `reducer` by `combineReducers` is a mutable object, the `reducer` can modify it directly and return it.

If it does, the shallow equality check that `combineReducers` performs will always pass, as the values of the `state` slice returned by the `reducer` may have been mutated, but the object itself has not - it's still the same object that was passed to the `reducer`.

Accordingly, `combineReducers` will not set its `hasChanged` flag, even though the `state` has changed. If none of the other reducers return a new, updated `state` slice, the `hasChanged` flag will remain set to false, causing `combineReducers` to return the *existing* root `state` object.

The `store` will still be updated with the new values for the root `state`, but because the root `state` object itself is still the same object, libraries that bind to Redux, such as React-Redux, will not be aware of the `state`'s mutation, and so will not trigger a wrapped component's re-rendering.

Further Information

Documentation

- [Recipes: Immutable Update Patterns](#)
- [Troubleshooting: Never mutate reducer arguments](#)

Why does a reducer mutating the state prevent React-Redux from re-rendering a wrapped component?

If a Redux [reducer](#) directly mutates, and returns, the [state](#) object passed into it, the values of the root [state](#) object will change, but the object itself will not.

Because React-Redux performs a shallow check on the root [state](#) object to determine if its wrapped components need re-rendering or not, it will not be able to detect the [state](#) mutation, and so will not trigger a re-rendering.

Further Information

Documentation

- [Troubleshooting: My views aren't updating when something changes outside of Redux](#)

Why does a selector mutating and returning a persistent object to `mapStateToProps` prevent React-Redux from re-rendering a wrapped component?

If one of the values of the props object returned from `mapStateToProps` is an object that persists across calls to `connect` (such as, potentially, the root [state](#) object), yet is directly mutated and returned by a selector function, React-Redux will not be able to detect the mutation, and so will not trigger a re-render of the wrapped component.

As we've seen, the values in the mutable object returned by the selector function may have changed, but the object itself has not, and shallow equality checking only compares the objects themselves, not their values.

For example, the following `mapStateToProps` function will never trigger a re-render:

```
// State object held in the Redux store
const state = {
  user: {
    accessCount: 0,
    name: 'keith'
  }
}

// Selector function
const getUser = state => {
  ++state.user.accessCount // mutate the state object
  return state
}

// mapStateToProps
const mapStateToProps = state => ({
  // The object returned from getUser() is always
  // the same object, so this wrapped
  // component will never re-render, even though it's been
  // mutated
  userRecord: getUser(state)
})

const a = mapStateToProps(state)
```

```
const b = mapStateToProps(state)

a.userRecord === b.userRecord
//> true
```

Note that, conversely, if an *immutable* object is used, the [component may re-render when it should not](#).

Further Information

Articles

- [Practical Redux, Part 6: Connected Lists, Forms, and Performance](#)

Discussions

- [#1948: Is getMappedItems an anti-pattern in mapStateToProps?](#)

How does immutability enable a shallow check to detect object mutations?

If an object is immutable, any changes that need to be made to it within a function must be made to a *copy* of the object.

This mutated copy is a *separate* object from that passed into the function, and so when it is returned, a shallow check will identify it as being a different object from that passed in, and so will fail.

Further Information

Articles

- [Pros and Cons of using immutability with React.js](#)

How can immutability in your reducers cause components to render unnecessarily?

You cannot mutate an immutable object; instead, you must mutate a copy of it, leaving the original intact.

That's perfectly OK when you mutate the copy, but in the context of a [reducer](#), if you return a copy that *hasn't* been mutated, Redux's `combineReducers` function will still think that the [state](#) needs to be updated, as you're returning an entirely different object from the [state](#) slice object that was passed in.

`combineReducers` will then return this new root [state](#) object to the [store](#). The new object will have the same values as the current root [state](#) object, but because it's a different object, it will cause the [store](#) to be updated, which will ultimately cause all connected components to be re-rendered unnecessarily.

To prevent this from happening, you must *always return the [state](#) slice object that's passed into a [reducer](#) if the [reducer](#) does not mutate the [state](#)*.

Further Information

Articles

- [React.js pure render performance anti-pattern](#)
- [Building Efficient UI with React and Redux](#)

How can immutability in `mapStateToProps` cause components to render unnecessarily?

Certain immutable operations, such as an Array filter, will always return a new object, even if the values themselves have not changed.

If such an operation is used as a selector function in `mapStateToProps`, the shallow equality check that React-Redux performs on each value in the props object that's returned will always fail, as the selector is returning a new object each time.

As such, even though the values of that new object have not changed, the wrapped component will always be re-rendered,

For example, the following will always trigger a re-render:

```
// A JavaScript array's 'filter' method treats the array as immutable,
// and returns a filtered copy of the array.
const getVisibleTodos = todos => todos.filter(t => !t.completed)

const state = {
  todos: [
    {
      text: 'do todo 1',
      completed: false
    },
    {
      text: 'do todo 2',
      completed: true
    }
  ]
}

const mapStateToProps = state => ({
  // getVisibleTodos() always returns a new array, and so the
  // 'visibleTodos' prop will always reference a different array,
  // causing the wrapped component to re-render, even if the array's
  // values haven't changed
  visibleTodos: getVisibleTodos(state.todos)
})

const a = mapStateToProps(state)
// Call mapStateToProps(state) again with exactly the same arguments
const b = mapStateToProps(state)

a.visibleTodos
//> { "completed": false, "text": "do todo 1" }

b.visibleTodos
//> { "completed": false, "text": "do todo 1" }

a.visibleTodos === b.visibleTodos
//> false
```

Note that, conversely, if the values in your props object refer to mutable objects, [your component may not render when it should](#).

Further Information

Articles

- [React.js pure render performance anti-pattern](#)
- [Building Efficient UI with React and Redux](#)
- [ImmutableJS: worth the price?](#)

What approaches are there for handling data immutability? Do I have to use Immutable.JS?

You do not need to use Immutable.JS with Redux. Plain JavaScript, if written correctly, is perfectly capable of providing immutability without having to use an immutable-focused library.

However, guaranteeing immutability with JavaScript is difficult, and it can be easy to mutate an object accidentally, causing bugs in your app that are extremely difficult to locate. For this reason, using an immutable update utility library such as Immutable.JS can significantly improve the reliability of your app, and make your app's development much easier.

Further Information

Discussions

- [#1185: Question: Should I use immutable data structures?](#)
- [Introduction to Immutable.js and Functional Programming Concepts](#)

What are the issues with using plain JavaScript for immutable operations?

JavaScript was never designed to provide guaranteed immutable operations. Accordingly, there are several issues you need to be aware of if you choose to use it for your immutable operations in your Redux app.

Accidental Object Mutation

With JavaScript, you can accidentally mutate an object (such as the Redux [state](#) tree) quite easily without realizing it. For example, updating deeply nested properties, creating a new *reference* to an object instead of a new object, or performing a shallow copy rather than a deep copy, can all lead to inadvertent object mutations, and can trip up even the most experienced JavaScript coder.

To avoid these issues, ensure you follow the recommended [immutable update patterns for ES6](#).

Verbose Code

Updating complex nested [state](#) trees can lead to verbose code that is tedious to write and difficult to debug.

Poor Performance

Operating on JavaScript objects and arrays in an immutable way can be slow, particularly as your [state](#) tree grows larger.

Remember, to change an immutable object, you must mutate a *copy* of it, and copying large objects can be slow as every property must be copied.

In contrast, immutable libraries such as Immutable.js can employ sophisticated optimization techniques such as [structural sharing](#) , which effectively returns a new object that reuses much of the existing object being copied from.

For copying very large objects, [plain JavaScript can be over 100 times slower](#) than an optimized immutable library.

Further Information

Documentation

- [Immutable Update Patterns for ES6](#)

Articles

- [Immutable.js, persistent data structures and structural sharing](#)
- [A deep dive into Clojure's data structures](#)
- [Introduction to Immutable.js and Functional Programming Concepts](#)
- [JavaScript and Immutability](#)
- [Immutable Javascript using ES6 and beyond](#)
- [Pros and Cons of using immutability with React.js - React Kung Fu](#)

Redux FAQ: Code Structure

Table of Contents

- [What should my file structure look like? How should I group my action creators and reducers in my project? Where should my selectors go?](#)
- [How should I split my logic between reducers and action creators? Where should my “business logic” go?](#)
- [Why should I use action creators?](#)
- [Where should websockets and other persistent connections live?](#)

Code Structure

What should my file structure look like? How should I group my action creators and reducers in my project? Where should my selectors go?

Since Redux is just a data [store](#) library, it has no direct opinion on how your project should be structured. However, there are a few common patterns that most Redux developers tend to use:

- Rails-style: separate folders for “actions”, “constants”, “reducers”, “containers”, and “components”
- Domain-style: separate folders per feature or domain, possibly with sub-folders per file type
- “Ducks”: similar to domain style, but explicitly tying together actions and reducers, often by defining them in the same file

It's generally suggested that selectors are defined alongside reducers and exported, and then reused elsewhere (such as in `mapStateToProps` functions, in async [action](#) creators, or sagas) to colocate all the code that knows about the actual shape of the [state](#) tree in the [reducer](#) files.

While it ultimately doesn't matter how you lay out your code on disk, it's important to remember that actions and reducers shouldn't be considered in isolation. It's entirely possible (and encouraged) for a [reducer](#) defined in one folder to respond to an [action](#) defined in another folder.

Further information

Documentation

- [FAQ: Actions - "1:1 mapping between reducers and actions?"](#)

Articles

- [How to Scale React Applications](#) (accompanying talk: [Scaling React Applications](#))
- [Redux Best Practices](#)
- [Rules For Structuring \(Redux\) Applications](#)
- [A Better File Structure for React/Redux Applications](#)
- [Organizing Large React Applications](#)
- [Four Strategies for Organizing Code](#)
- [Encapsulating the Redux State Tree](#)
- [Redux Reducer/Selector Asymmetry](#)
- [Modular Reducers and Selectors](#)

- [My journey towards a maintainable project structure for React/Redux](#)
- [React/Redux Links: Architecture - Project File Structure](#)

Discussions

- [#839: Emphasize defining selectors alongside reducers](#)
- [#943: Reducer querying](#)
- [React Boilerplate #27: Application Structure](#)
- [Stack Overflow: How to structure Redux components/containers](#)
- [Twitter: There is no ultimate file structure for Redux](#)

How should I split my logic between reducers and action creators? Where should my “business logic” go?

There's no single clear answer to exactly what pieces of logic should go in a [reducer](#) or an [action](#) creator. Some developers prefer to have “fat” [action](#) creators, with “thin” reducers that simply take the data in an [action](#) and blindly merge it into the corresponding [state](#). Others try to emphasize keeping actions as small as possible, and minimize the usage of `getState()` in an [action](#) creator. (For purposes of this question, other async approaches such as sagas and observables fall in the “[action](#) creator” category.)

There are some potential benefits from putting more logic into your reducers. It's likely that the [action](#) types would be more semantic and more meaningful (such as `"USER_UPDATED"` instead of `"SET_STATE"`). In addition, having more logic in reducers means that more functionality will be affected by time travel debugging.

This comment sums up the dichotomy nicely:

Now, the problem is what to put in the [action](#) creator and what in the [reducer](#), the choice between fat and thin [action](#) objects. If you put all the logic in the [action](#) creator, you end up with fat [action](#) objects that basically declare the updates to the [state](#). Reducers become pure, dumb, add-this, remove that, update these functions. They will be easy to compose. But not much of your business logic will be there. If you put more logic in the [reducer](#), you end up with nice, thin [action](#) objects, most of your data logic in one place, but your reducers are harder to compose since you might need info from other branches. You end up with large reducers or reducers that take additional arguments from higher up in the [state](#).

Find the balance between these two extremes, and you will master Redux.

Further information

Articles

- [Where do I put my business logic in a React/Redux application?](#)
- [How to Scale React Applications](#)
- [The Tao of Redux, Part 2 - Practice and Philosophy. Thick and thin reducers.](#)

Discussions

- [How putting too much logic in action creators could affect debugging](#)
- [#384: The more that's in a reducer, the more you can replay via time travel](#)
- [#1165: Where to put business logic / validation?](#)
- [#1171: Recommendations for best practices regarding action-creators, reducers, and selectors](#)
- [Stack Overflow: Accessing Redux state in an action creator?](#)
- [#2796: Gaining clarity on "business logic"](#)
- [Twitter: Moving away from unclear terminology...](#)

Why should I use action creators?

Redux does not require [action](#) creators. You are free to create actions in any way that is best for you, including simply passing an object literal to `dispatch`. [Action](#) creators emerged from the [Flux architecture](#) and have been adopted by the Redux community because they offer several benefits.

[Action](#) creators are more maintainable. Updates to an [action](#) can be made in one place and applied everywhere. All instances of an [action](#) are guaranteed to have the same shape and the same default values.

[Action](#) creators are testable. The correctness of an inline [action](#) must be verified manually. Like any function, tests for an [action](#) creator can be written once and run automatically.

[Action](#) creators are easier to document. The [action](#) creator's parameters enumerate the [action](#)'s dependencies. And centralization of the [action](#) definition provides a convenient place for documentation comments. When actions are written inline, this information is harder to capture and communicate.

[Action](#) creators are a more powerful abstraction. Creating an [action](#) often involves transforming data or making AJAX requests. [Action](#) creators provide a uniform interface to this varied logic. This abstraction frees a component to dispatch an [action](#) without being complicated by the details of that [action](#)'s creation.

Further information

Articles

- [Idiomatic Redux: Why use action creators?](#)

Discussions

- [Reddit: Redbox - Redux action creation made simple](#)

Where should websockets and other persistent connections live?

[Middleware](#) are the right place for persistent connections like websockets in a Redux app, for several reasons:

- [Middleware](#) exist for the lifetime of the application
- Like with the [store](#) itself, you probably only need a single instance of a given connection that the whole app can use
- [Middleware](#) can see all dispatched actions and dispatch actions themselves. This means a [middleware](#) can take dispatched actions and turn those into messages sent over the websocket, and dispatch new actions when a message is received over the websocket.
- A websocket connection instance isn't serializable, so [it doesn't belong in the store state itself](#)

See [this example that shows how a socket middleware might dispatch and respond to Redux actions](#).

There's many existing [middleware](#) for websockets and other similar connections - see the link below.

Libraries

- [Middleware: Socket and Adapters](#)

Redux FAQ: Performance

Table of Contents

- [How well does Redux “scale” in terms of performance and architecture?](#)
- [Won't calling “all my reducers” for each action be slow?](#)
- [Do I have to deep-clone my state in a reducer? Isn't copying my state going to be slow?](#)
- [How can I reduce the number of store update events?](#)
- [Will having “one state tree” cause memory problems? Will dispatching many actions take up memory?](#)
- [Will caching remote data cause memory problems?](#)

Performance

How well does Redux “scale” in terms of performance and architecture?

While there's no single definitive answer to this, most of the time this should not be a concern in either case.

The work done by Redux generally falls into a few areas: processing actions in [middleware](#) and reducers (including object duplication for immutable updates), notifying subscribers after actions are dispatched, and updating UI components based on the [state](#) changes. While it's certainly *possible* for each of these to become a performance concern in sufficiently complex situations, there's nothing inherently slow or inefficient about how Redux is implemented. In fact, React Redux in particular is heavily optimized to cut down on unnecessary re-renders, and React-Redux v5 shows noticeable improvements over earlier versions.

Redux may not be as efficient out of the box when compared to other libraries. For maximum rendering performance in a React application, [state](#) should be stored in a normalized shape, many individual components should be connected to the [store](#) instead of just a few, and connected list components should pass item IDs to their connected child list items (allowing the list items to look up their own data by ID). This minimizes the overall amount of rendering to be done. Use of memoized selector functions is also an important performance consideration.

As for architecture, anecdotal evidence is that Redux works well for varying project and team sizes. Redux is currently used by hundreds of companies and thousands of developers, with several hundred thousand monthly installations from NPM. One developer reported:

for scale, we have ~500 [action](#) types, ~400 [reducer](#) cases, ~150 components, 5 middlewares, ~200 actions, ~2300 tests

Further information

Documentation

- [Recipes: Structuring Reducers - Normalizing State Shape](#)

Articles

- [How to Scale React Applications](#) (accompanying talk: [Scaling React Applications](#))
- [High-Performance Redux](#)
- [Improving React and Redux Perf with Reselect](#)

- [Encapsulating the Redux State Tree](#)
- [React/Redux Links: Performance - Redux](#)

Discussions

- [#310: Who uses Redux?](#)
- [#1751: Performance issues with large collections](#)
- [React Redux #269: Connect could be used with a custom subscribe method](#)
- [React Redux #407: Rewrite connect to offer an advanced API](#)
- [React Redux #416: Rewrite connect for better performance and extensibility](#)
- [Redux vs MobX TodoMVC Benchmark: #1](#)
- [Reddit: What's the best place to keep the initial state?](#)
- [Reddit: Help designing Redux state for a single page app](#)
- [Reddit: Redux performance issues with a large state object?](#)
- [Reddit: React/Redux for Ultra Large Scale apps](#)
- [Twitter: Redux scaling](#)
- [Twitter: Redux vs MobX benchmark graph - Redux state shape matters](#)
- [Stack Overflow: How to optimize small updates to props of nested components?](#)
- [Chat log: React/Redux perf - updating a 10K-item Todo list](#)
- [Chat log: React/Redux perf - single connection vs many connections](#)

Won't calling “all my reducers” for each action be slow?

It's important to note that a Redux [store](#) really only has a single [reducer](#) function. The [store](#) passes the current [state](#) and dispatched [action](#) to that one [reducer](#) function, and lets the [reducer](#) handle things appropriately.

Obviously, trying to handle every possible [action](#) in a single function does not scale well, simply in terms of function size and readability, so it makes sense to split the actual work into separate functions that can be called by the top-level [reducer](#). In particular, the common suggested pattern is to have a separate sub-[reducer](#) function that is responsible for managing updates to a particular slice of [state](#) at a specific key. The `combineReducers()` that comes with Redux is one of the many possible ways to achieve this. It's also highly suggested to keep your [store state](#) as flat and as normalized as possible. Ultimately, though, you are in charge of organizing your [reducer](#) logic any way you want.

However, even if you happen to have many different [reducer](#) functions composed together, and even with deeply nested [state](#), [reducer](#) speed is unlikely to be a problem. JavaScript engines are capable of running a very large number of function calls per second, and most of your reducers are probably just using a `switch` statement and returning the existing [state](#) by default in response to most actions.

If you actually are concerned about [reducer](#) performance, you can use a utility such as [redux-ignore](#) or [redux-scoped-reducer](#) to ensure that only certain reducers listen to specific actions. You can also use [redux-log-slow-reducers](#) to do some performance benchmarking.

Further information

Discussions

- [#912: Proposal: action filter utility](#)
- [#1303: Redux Performance with Large Store and frequent updates](#)
- [Stack Overflow: State in Redux app has the name of the reducer](#)
- [Stack Overflow: How does Redux deal with deeply nested models?](#)

Do I have to deep-clone my state in a reducer? Isn't copying my state going to be slow?

Immutably updating [state](#) generally means making shallow copies, not deep copies. Shallow copies are much faster than deep copies, because fewer objects and fields have to be copied, and it effectively comes down to moving some pointers around.

In addition, deep cloning [state](#) creates new references for every field. Since the React-Redux `connect` function relies on reference comparisons to determine if data has changed, this means that UI components will be forced to re-render unnecessarily even though the other data hasn't meaningfully changed.

However, you *do* need to create a copied and updated object for each level of nesting that is affected. Although that shouldn't be particularly expensive, it's another good reason why you should keep your [state](#) normalized and shallow if possible.

Common Redux misconception: you need to deeply clone the [state](#). Reality: if something inside doesn't change, keep its reference the same!

Further information

Documentation

- [Recipes: Structuring Reducers - Prerequisite Concepts](#)
- [Recipes: Structuring Reducers - Immutable Update Patterns](#)

Discussions

- [#454: Handling big states in reducer](#)
- [#758: Why can't state be mutated?](#)
- [#994: How to cut the boilerplate when updating nested entities?](#)
- [Twitter: common misconception - deep cloning](#)
- [Cloning Objects in JavaScript](#)

How can I reduce the number of store update events?

Redux notifies subscribers after each successfully dispatched [action](#) (i.e. an [action](#) reached the [store](#) and was handled by reducers). In some cases, it may be useful to cut down on the number of times subscribers are called, particularly if an [action](#) creator dispatches multiple distinct actions in a row.

There are several addons that add batching capabilities in various ways, like: [redux-batched-actions](#) (a higher-order [reducer](#) that lets you dispatch several actions as if it was one and “unpack” them in the [reducer](#)), [redux-batched-subscribe](#) (a [store](#) enhancer that lets you debounce subscriber calls for multiple dispatches), or [redux-batch](#) (a [store](#) enhancer that handles dispatching an array of actions with a single subscriber notification).

For React-Redux specifically, starting in [React-Redux v7](#) a new `batch` public API is available to help minimize the number of React re-renders when dispatching actions outside of React event handlers. It wraps React's `unstable_batchedUpdate()` API, allows any React updates in an event loop tick to be batched together into a single render pass. React already uses this internally for its own event handler callbacks. This API is actually part of the renderer packages like ReactDOM and React Native, not the React core itself.

Since React-Redux needs to work in both ReactDOM and React Native environments, we've taken care of importing this API from the correct renderer at build time for our own use. We also now re-export this function publicly ourselves, renamed to `batch()`. You can use it to ensure that multiple actions dispatched outside of

React only result in a single render update, like this:

```
import { batch } from "react-redux";

function myThunk() {
  return (dispatch, getState) => {
    // should only result in one combined re-render, not two
    batch(() => {
      dispatch(increment());
      dispatch(increment());
    })
  }
}
```

Further information

Discussions

- [#125: Strategy for avoiding cascading renders](#)
- [#542: Idea: batching actions](#)
- [#911: Batching actions](#)
- [#1813: Use a loop to support dispatching arrays](#)
- [React Redux #263: Huge performance issue when dispatching hundreds of actions](#)
- [React-Redux #1177: Roadmap: v6, Context, Subscriptions, and Hooks](#)

Libraries

- [Redux Addons Catalog: Store - Change Subscriptions](#)

Will having “one state tree” cause memory problems? Will dispatching many actions take up memory?

First, in terms of raw memory usage, Redux is no different than any other JavaScript library. The only difference is that all the various object references are nested together into one tree, instead of maybe saved in various independent model instances such as in Backbone. Second, a typical Redux app would probably have somewhat *less* memory usage than an equivalent Backbone app because Redux encourages use of plain JavaScript objects and arrays rather than creating instances of Models and Collections. Finally, Redux only holds onto a single [state](#) tree reference at a time. Objects that are no longer referenced in that tree will be garbage collected, as usual.

Redux does not [store](#) a history of actions itself. However, the Redux DevTools do [store](#) actions so they can be replayed, but those are generally only enabled during development, and not used in production.

Further information

Documentation

- [Docs: Async Actions](#)

Discussions

- [Stack Overflow: Is there any way to "commit" the state in Redux to free memory?](#)
- [Stack Overflow: Can a Redux store lead to a memory leak?](#)
- [Stack Overflow: Redux and ALL the application state](#)
- [Stack Overflow: Memory Usage Concern with Controlled Components](#)

- [Reddit: What's the best place to keep initial state?](#)

Will caching remote data cause memory problems?

The amount of memory available to JavaScript applications running in a browser is finite. Caching data will cause performance problems when the size of the cache approaches the amount of available memory. This tends to be a problem when the cached data is exceptionally large or the session is exceptionally long-running. And while it is good to be aware of the potential for these problems, this awareness should not discourage you from efficiently caching reasonable amounts of data.

Here are a few approaches to caching remote data efficiently:

First, only cache as much data as the user needs. If your application displays a paginated list of records, you don't necessarily need to cache the entire collection. Instead, cache what is visible to the user and add to that cache when the user has (or will soon have) an immediate need for more data.

Second, cache an abbreviated form of a record when possible. Sometimes a record includes data that is not relevant to the user. If the application does not depend on this data, it can be omitted from the cache.

Third, only cache a single copy of a record. This is especially important when records contain copies of other records. Cache a unique copy for each record and replace each nested copy with a reference. This is called normalization. Normalization is the preferred approach to storing relational data for [several reasons](#), including efficient memory consumption.

Further information

Discussions

- [Stack Overflow: How to choose the Redux state shape for an app with list/detail views and pagination?](#)
- [Twitter: ...concerns over having "too much data in the state tree"...](#)
- [Advanced Redux entity normalization](#)

Redux FAQ: Design Decisions

Table of Contents

- [Why doesn't Redux pass the state and action to subscribers?](#)
- [Why doesn't Redux support using classes for actions and reducers?](#)
- [Why does the middleware signature use currying?](#)
- [Why does applyMiddleware use a closure for dispatch?](#)
- [Why doesn't `combineReducers` include a third argument with the entire state when it calls each reducer?](#)
- [Why doesn't `mapDispatchToProps` allow use of return values from `getState\(\)` or `mapStateToProps`?](#)

Design Decisions

Why doesn't Redux pass the state and action to subscribers?

Subscribers are intended to respond to the [state](#) value itself, not the [action](#). Updates to the [state](#) are processed synchronously, but notifications to subscribers can be batched or debounced, meaning that subscribers are not always notified with every [action](#). This is a common [performance optimization](#) to avoid repeated re-rendering.

Batching or debouncing is possible by using enhancers to override `store.dispatch` to change the way that subscribers are notified. Also, there are libraries that change Redux to process actions in batches to optimize performance and avoid repeated re-rendering:

- [redux-batch](#) allows passing an array of actions to `store.dispatch()` with only one notification,
- [redux-batched-subscribe](#) allows batching of subscribe notifications that occur as a result of dispatches.

The intended guarantee is that Redux eventually calls all subscribers with the most recent [state](#) available, but not that it always calls each subscriber for each [action](#). The [store state](#) is available in the subscriber simply by calling `store.getState()`. The [action](#) cannot be made available in the subscribers without breaking the way that actions might be batched.

A potential use-case for using the [action](#) inside a subscriber -- which is an unsupported feature -- is to ensure that a component only re-renders after certain kinds of actions. Instead, re-rendering should be controlled through:

1. the [shouldComponentUpdate](#) lifecycle method
2. the [virtual DOM equality check \(vDOMEq\)](#)
3. [React.PureComponent](#)
4. Using React-Redux: use [mapStateToProps](#) to subscribe components to only the parts of the [store](#) that they need.

Further Information

Articles

- [How can I reduce the number of store update events?](#)

Discussions

- [#580: Why doesn't Redux pass the state to subscribers?](#)

- [#2214: Alternate Proof of Concept: Enhancer Overhaul -- more on debouncing](#)

Why doesn't Redux support using classes for actions and reducers?

The pattern of using functions, called [action](#) creators, to return [action](#) objects may seem counterintuitive to programmers with a lot of Object Oriented Programming experience, who would see this is a strong use-case for Classes and instances. Class instances for [action](#) objects and reducers are not supported because class instances make serialization and deserialization tricky. Deserialization methods like `JSON.parse(string)` will return a plain old Javascript object rather than class instances.

As described in the [Store FAQ](#), if you are okay with things like persistence and time-travel debugging not working as intended, you are welcome to put non-serializable items into your Redux [store](#).

Serialization enables the browser to [store](#) all actions that have been dispatched, as well as the previous [store](#) states, with much less memory. Rewinding and 'hot reloading' the [store](#) is central to the Redux developer experience and the function of Redux DevTools. This also enables deserialized actions to be stored on the server and re-serialized in the browser in the case of server-side rendering with Redux.

Further Information

Articles

- [Can I put functions, promises, or other non-serializable items in my store state?](#)

Discussions

- [#1171: Why doesn't Redux use classes for actions and reducers?](#)

Why does the middleware signature use currying?

Redux [middleware](#) are written using a triply-nested function structure that looks like `const middleware = storeAPI => next => action => {}`, rather than a single function that looks like `const middleware = (storeAPI, next, action) => {}`. There's a few reasons for this.

One is that "currying" functions is a standard functional programming technique, and Redux was explicitly intended to use functional programming principles in its design. Another is that currying functions creates closures where you can declare variables that exist for the lifetime of the [middleware](#) (which could be considered a functional equivalent to instance variables that exist for the lifetime of a class instance). Finally, it's simply the approach that was chosen when Redux was initially designed.

The [curried function signature](#) of declaring [middleware](#) is [deemed unnecessary](#) by some, because both [store](#) and [next](#) are available when the `applyMiddleware` function is executed. This issue has been determined to not be [worth introducing breaking changes](#), as there are now hundreds of [middleware](#) in the Redux ecosystem that rely on the existing [middleware](#) definition.

Further Information

Discussions

- Why does the [middleware](#) signature use currying?
 - Prior discussions: [#55](#), [#534](#), [#784](#), [#922](#), [#1744](#)
 - [React Boston 2017: You Might Need Redux \(And Its Ecosystem\)](#)

Why does `applyMiddleware` use a closure for `dispatch` ?

`applyMiddleware` takes the existing dispatch from the `store` and closes over it to create the initial chain of middlewares that have been invoked with an object that exposes the `getState` and `dispatch` functions, which enables middlewares that [rely on dispatch during initialization](#) to run.

Further Information

Discussions

- Why does `applyMiddleware` use a closure for `dispatch`?
 - See - [#1592](#) and [#2097](#)

Why doesn't `combineReducers` include a third argument with the entire state when it calls each reducer?

`combineReducers` is opinionated to encourage splitting `reducer` logic by domain. As stated in [Beyond `combineReducers`](#), `combineReducers` is deliberately limited to handle a single common use case: updating a `state` tree that is a plain Javascript object by delegating the work of updating each slice of `state` to a specific slice `reducer`.

It's not immediately obvious what a potential third argument to each `reducer` should be: the entire `state` tree, some callback function, some other part of the `state` tree, etc. If `combineReducers` doesn't fit your use case, consider using libraries like [combineSectionReducers](#) or [reduceReducers](#) for other options with deeply nested reducers and reducers that require access to the global `state`.

If none of the published utilities solve your use case, you can always write a function yourself that does just exactly what you need.

Further information

Articles

- [Beyond `combineReducers`](#)

Discussions

- [#1768 Allow reducers to consult global state](#)

Why doesn't `mapDispatchToProps` allow use of return values from `getState()` or `mapStateToProps()` ?

There have been requests to use either the entire `state` or the return value of `mapState` inside of `mapDispatch`, so that when functions are declared inside of `mapDispatch`, they can close over the latest returned values from the `store`.

This approach is not supported in `mapDispatch` because it would mean also calling `mapDispatch` every time the `store` is updated. This would cause the re-creation of functions with every `state` update, thus adding a lot of performance overhead.

The preferred way to handle this use-case--needing to alter props based on the current `state` and `mapDispatchToProps` functions--is to work from `mergeProps`, the third argument to the `connect` function. If specified, it is passed the result of `mapStateToProps()`, `mapDispatchToProps()`, and the container component's props. The plain object returned from `mergeProps` will be passed as props to the wrapped component.

Further information

Discussions

- [#237 Why doesn't mapDispatchToProps allow use of return values from getState\(\) or mapStateToProps\(\)](#)?

Redux FAQ: React Redux

Table of Contents

- [Why should I use React-Redux?](#)
- [Why isn't my component re-rendering, or my mapStateToProps running?](#)
- [Why is my component re-rendering too often?](#)
- [How can I speed up my mapStateToProps?](#)
- [Why don't I have this.props.dispatch available in my connected component?](#)
- [Should I only connect my top component, or can I connect multiple components in my tree?](#)
- [How does Redux compare to the React Context API?](#)

React Redux

Why should I use React-Redux?

Redux itself is a standalone library that can be used with any UI layer or framework, including React, Angular, Vue, Ember, and vanilla JS. Although Redux and React are commonly used together, they are independent of each other.

If you are using Redux with any kind of UI framework, you will normally use a "UI binding" library to tie Redux together with your UI framework, rather than directly interacting with the [store](#) from your UI code.

React-Redux is the official Redux UI binding library for React. If you are using Redux and React together, you should also use React-Redux to bind these two libraries.

While it is possible to write Redux [store](#) subscription logic by hand, doing so would become very repetitive. In addition, optimizing UI performance would require complicated logic.

The process of subscribing to the [store](#), checking for updated data, and triggering a re-render can be made more generic and reusable. **A UI binding library like React-Redux handles the [store](#) interaction logic, so you don't have to write that code yourself.**

Overall, React-Redux encourages good React architecture, and implements complex performance optimizations for you. It is also kept up-to-date with the latest API changes from Redux and React.

Further Information

Documentation

- [React-Redux docs: Why Use React-Redux?](#)

Why isn't my component re-rendering, or my mapStateToProps running?

Accidentally mutating or modifying your [state](#) directly is by far the most common reason why components do not re-render after an [action](#) has been dispatched. Redux expects that your reducers will update their [state](#) "immutably", which effectively means always making copies of your data, and applying your changes to the copies. If you return the same object from a [reducer](#), Redux assumes that nothing has been changed, even if you

made changes to its contents. Similarly, React Redux tries to improve performance by doing shallow equality reference checks on incoming props in `shouldComponentUpdate`, and if all references are the same, `shouldComponentUpdate` returns `false` to skip actually updating your original component.

It's important to remember that whenever you update a nested value, you must also return new copies of anything above it in your `state` tree. If you have `state.a.b.c.d`, and you want to make an update to `d`, you would also need to return new copies of `c`, `b`, `a`, and `state`. This [state tree mutation diagram](#) demonstrates how a change deep in a tree requires changes all the way up.

Note that “updating data immutably” does *not* mean that you must use [Immutable.js](#), although that is certainly an option. You can do immutable updates to plain JS objects and arrays using several different approaches:

- Copying objects using functions like `Object.assign()` or `_.extend()`, and array functions such as `slice()` and `concat()`
- The array spread operator in ES6, and the similar object spread operator that is proposed for a future version of JavaScript
- Utility libraries that wrap immutable update logic into simpler functions

Further information

Documentation

- [Troubleshooting](#)
- [React Redux: Troubleshooting](#)
- [Recipes: Using the Object Spread Operator](#)
- [Recipes: Structuring Reducers - Prerequisite Concepts](#)
- [Recipes: Structuring Reducers - Immutable Update Patterns](#)

Articles

- [Pros and Cons of Using Immutability with React](#)
- [React/Redux Links: Immutable Data](#)

Discussions

- [#1262: Immutable data + bad performance](#)
- [React Redux #235: Predicate function for updating component](#)
- [React Redux #291: Should `mapStateToProps` be called every time an action is dispatched?](#)
- [Stack Overflow: Cleaner/shorter way to update nested state in Redux?](#)
- [Gist: state mutations](#)

Why is my component re-rendering too often?

React Redux implements several optimizations to ensure your actual component only re-renders when actually necessary. One of those is a shallow equality check on the combined props object generated by the `mapStateToProps` and `mapDispatchToProps` arguments passed to `connect`. Unfortunately, shallow equality does not help in cases where new array or object instances are created each time `mapStateToProps` is called. A typical example might be mapping over an array of IDs and returning the matching object references, such as:

```
const mapStateToProps = state => {
  return {
    objects: state.objectIds.map(id => state.objects[id])
  }
}
```

Even though the array might contain the exact same object references each time, the array itself is a different reference, so the shallow equality check fails and React Redux would re-render the wrapped component.

The extra re-renders could be resolved by saving the array of objects into the [state](#) using a [reducer](#), caching the mapped array using [Reselect](#), or implementing `shouldComponentUpdate` in the component by hand and doing a more in-depth props comparison using a function such as `_.isEqual`. Be careful to not make your custom `shouldComponentUpdate()` more expensive than the rendering itself! Always use a profiler to check your assumptions about performance.

For non-connected components, you may want to check what props are being passed in. A common issue is having a parent component re-bind a callback inside its render function, like `<Child onClick={this.handleClick.bind(this)} />`. That creates a new function reference every time the parent re-renders. It's generally good practice to only bind callbacks once in the parent component's constructor.

Further information

Documentation

- [FAQ: Performance - Scaling](#)

Articles

- [A Deep Dive into React Perf Debugging](#)
- [React.js pure render performance anti-pattern](#)
- [Improving React and Redux Performance with Reselect](#)
- [Encapsulating the Redux State Tree](#)
- [React/Redux Links: React/Redux Performance](#)

Discussions

- [Stack Overflow: Can a React Redux app scale as well as Backbone?](#)

Libraries

- [Redux Addons Catalog: DevTools - Component Update Monitoring](#)

How can I speed up my `mapStateToProps` ?

While React Redux does work to minimize the number of times that your `mapStateToProps` function is called, it's still a good idea to ensure that your `mapStateToProps` runs quickly and also minimizes the amount of work it does. The common recommended approach is to create memoized “selector” functions using [Reselect](#). These selectors can be combined and composed together, and selectors later in a pipeline will only run if their inputs have changed. This means you can create selectors that do things like filtering or sorting, and ensure that the real work only happens if needed.

Further information

Documentation

- [Recipes: Computed Derived Data](#)

Articles

- [Improving React and Redux Performance with Reselect](#)

Discussions

- [#815: Working with Data Structures](#)
- [Reselect #47: Memoizing Hierarchical Selectors](#)

Why don't I have `this.props.dispatch` available in my connected component?

The `connect()` function takes two primary arguments, both optional. The first, `mapStateToProps`, is a function you provide to pull data from the `store` when it changes, and pass those values as props to your component. The second, `mapDispatchToProps`, is a function you provide to make use of the `store`'s `dispatch` function, usually by creating pre-bound versions of `action` creators that will automatically dispatch their actions as soon as they are called.

If you do not provide your own `mapDispatchToProps` function when calling `connect()`, React Redux will provide a default version, which simply returns the `dispatch` function as a prop. That means that if you *do* provide your own function, `dispatch` is *not* automatically provided. If you still want it available as a prop, you need to explicitly return it yourself in your `mapDispatchToProps` implementation.

Further information

Documentation

- [React Redux API: connect\(\)](#)

Discussions

- [React Redux #89: can i wrap multi actionCreators into one props with name?](#)
- [React Redux #145: consider always passing down dispatch regardless of what mapDispatchToProps does](#)
- [React Redux #255: this.props.dispatch is undefined if using mapDispatchToProps](#)
- [Stack Overflow: How to get simple dispatch from this.props using connect w/ Redux?](#)

Should I only connect my top component, or can I connect multiple components in my tree?

Early Redux documentation advised that you should only have a few connected components near the top of your component tree. However, time and experience has shown that such a component architecture generally requires a few components to know too much about the data requirements of all their descendants, and forces them to pass down a confusing number of props.

The current suggested best practice is to categorize your components as “presentational” or “container” components, and extract a connected container component wherever it makes sense:

Emphasizing “one container component at the top” in Redux examples was a mistake. Don't take this as a maxim. Try to keep your presentation components separate. Create container components by connecting them when it's convenient. Whenever you feel like you're duplicating code in parent components to provide data for same kinds of children, time to extract a container. Generally as soon as you feel a parent knows too much about “personal” data or actions of its children, time to extract a container.

In fact, benchmarks have shown that more connected components generally leads to better performance than fewer connected components.

In general, try to find a balance between understandable data flow and areas of responsibility with your components.

Further information

Documentation

- [Basics: Usage with React](#)
- [FAQ: Performance - Scaling](#)

Articles

- [Presentational and Container Components](#)
- [High-Performance Redux](#)
- [React/Redux Links: Architecture - Redux Architecture](#)
- [React/Redux Links: Performance - Redux Performance](#)

Discussions

- [Twitter: emphasizing “one container” was a mistake](#)
- [#419: Recommended usage of connect](#)
- [#756: container vs component?](#)
- [#1176: Redux+React with only stateless components](#)
- [Stack Overflow: can a dumb component use a Redux container?](#)

How does Redux compare to the React Context API?

Similarities

Both Redux and React's Context API deal with "prop drilling". That said, they both allow you to pass data without having to pass the props through multiple layers of components. Internally, Redux *uses* the React context API that allows it to pass the [store](#) along your component tree.

Differences

With Redux, you get the the power of [Redux Dev Tools Extension](#). It automatically logs every [action](#) your app performs, and it allows time traveling – you can click on any past [action](#) and jump to that point in time. Redux also supports the concept of [middleware](#), where you may bind customized function calls on every [action](#) dispatch. Such examples include an automatic event logger, interception of certain actions, etc.

With React's Context API, you deal with a pair of components speaking only to each other. This gives you nice isolation between irrelevant data. You also have the flexibility of how you may use the data with your components, i.e., you can provide the [state](#) of a parent component, and you may pass context data as props to wrapped components.

There is a key difference in how Redux and React's Context treat data. Redux maintains the data of your whole app in a giant, stateful object. It deduces the changes of your data by running the [reducer](#) function you provide, and returns the next [state](#) that corresponds to every [action](#) dispatched. React Redux then optimizes component rendering and makes sure that each component re-renders only when the data it needs change. Context, on the other hand, does not hold any [state](#). It is only a conduit for the data. To express changes in data you need to rely on the [state](#) of a parent component.

Further information

- [Redux vs. The React Context API](#)
- [You Might Not Need Redux \(But You Can't Replace It With Hooks\)](#)

Redux FAQ: Miscellaneous

Table of Contents

- [Are there any larger, “real” Redux projects?](#)
- [How can I implement authentication in Redux?](#)

Miscellaneous

Are there any larger, “real” Redux projects?

Yes, lots of them! To name just a few:

- [Twitter's mobile site](#)
- [Wordpress's new admin page](#)
- [Firefox's new debugger](#)
- [The HyperTerm terminal application](#)

And many, many more! The Redux Addons Catalog has [a list of Redux-based applications and examples](#) that points to a variety of actual applications, large and small.

Further information

Documentation

- [Introduction: Examples](#)

Discussions

- [Reddit: Large open source react/redux projects?](#)
- [HN: Is there any huge web application built using Redux?](#)

How can I implement authentication in Redux?

Authentication is essential to any real application. When going about authentication you must keep in mind that nothing changes with how you should organize your application and you should implement authentication in the same way you would any other feature. It is relatively straightforward:

1. Create [action](#) constants for `LOGIN_SUCCESS` , `LOGIN_FAILURE` , etc.
2. Create [action](#) creators that take in credentials, a flag that signifies whether authentication succeeded, a token, or an error message as the payload.
3. Create an async [action](#) creator with Redux Thunk [middleware](#) or any [middleware](#) you see fit to fire a network request to an API that returns a token if the credentials are valid. Then save the token in the local storage or show a response to the user if it failed. You can perform these side effects from the [action](#) creators you wrote in the previous step.
4. Create a [reducer](#) that returns the next [state](#) for each possible authentication case (`LOGIN_SUCCESS` , `LOGIN_FAILURE` , etc).

Further information

Articles

- [Authentication with JWT by Auth0](#)
- [Tips to Handle Authentication in Redux](#)

Examples

- [react-redux-jwt-auth-example](#)

Libraries

- [Redux Addons Catalog: Use Cases - Authentication](#)

Troubleshooting

This is a place to share common problems and solutions to them.

The examples use React, but you should still find them useful if you use something else.

Nothing happens when I dispatch an action

Sometimes, you are trying to dispatch an [action](#), but your view does not update. Why does this happen? There may be several reasons for this.

Never mutate reducer arguments

It is tempting to modify the `state` or `action` passed to you by Redux. Don't do this!

Redux assumes that you never mutate the objects it gives to you in the [reducer](#). **Every single time, you must return the new `state` object.** Even if you don't use a library like [Immutable](#), you need to completely avoid mutation.

Immutability is what lets [react-redux](#) efficiently subscribe to fine-grained updates of your `state`. It also enables great developer experience features such as time travel with [redux-devtools](#).

For example, a [reducer](#) like this is wrong because it mutates the `state`:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // Wrong! This mutates state
      state.push({
        text: action.text,
        completed: false
      })
      return state
    case 'COMPLETE_TODO':
      // Wrong! This mutates state[action.index].
      state[action.index].completed = true
      return state
    default:
      return state
  }
}
```

It needs to be rewritten like this:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      // Return a new array
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      // Return a new array
```

```

    return state.map((todo, index) => {
      if (index === action.index) {
        // Copy the object before mutating
        return Object.assign({}, todo, {
          completed: true
        })
      }
      return todo
    })
  default:
    return state
  }
}

```

It's more code, but it's exactly what makes Redux predictable and efficient. If you want to have less code, you can use a helper like `React.addons.update` to write immutable transformations with a terse syntax:

```

// Before:
return state.map((todo, index) => {
  if (index === action.index) {
    return Object.assign({}, todo, {
      completed: true
    })
  }
  return todo
})

// After
return update(state, {
  [action.index]: {
    completed: {
      $set: true
    }
  }
})

```

Finally, to update objects, you'll need something like `_.extend` from Underscore, or better, an `Object.assign` polyfill.

Make sure that you use `Object.assign` correctly. For example, instead of returning something like `Object.assign(state, newData)` from your reducers, return `Object.assign({}, state, newData)`. This way you don't override the previous `state`.

You can also enable the [object spread operator proposal](#) for a more succinct syntax:

```

// Before:
return state.map((todo, index) => {
  if (index === action.index) {
    return Object.assign({}, todo, {
      completed: true
    })
  }
  return todo
})

// After:
return state.map((todo, index) => {
  if (index === action.index) {
    return { ...todo, completed: true }
  }
  return todo
})

```

```
}}
```

Note that experimental language features are subject to change.

Also keep an eye out for nested [state](#) objects that need to be deeply copied. Both `_.extend` and `Object.assign` make a shallow copy of the [state](#). See [Updating Nested Objects](#) for suggestions on how to deal with nested [state](#) objects.

Don't forget to call `dispatch(action)`

If you define an [action](#) creator, calling it will *not* automatically dispatch the [action](#). For example, this code will do nothing:

TodoActions.js

```
export function addTodo(text) {  
  return { type: 'ADD_TODO', text }  
}
```

AddTodo.js

```
import React, { Component } from 'react'  
import { addTodo } from './TodoActions'  
  
class AddTodo extends Component {  
  handleClick() {  
    // Won't work!  
    addTodo('Fix the issue')  
  }  
  
  render() {  
    return <button onClick={() => this.handleClick()}>Add</button>  
  }  
}
```

It doesn't work because your [action](#) creator is just a function that *returns* an [action](#). It is up to you to actually dispatch it. We can't bind your [action](#) creators to a particular [Store](#) instance during the definition because apps that render on the server need a separate Redux [store](#) for every request.

The fix is to call `dispatch()` method on the [store](#) instance:

```
handleClick() {  
  // Works! (but you need to grab store somehow)  
  store.dispatch(addTodo('Fix the issue'))  
}
```

If you're somewhere deep in the component hierarchy, it is cumbersome to pass the [store](#) down manually. This is why [react-redux](#) lets you use a `connect` [higher-order component](#) that will, apart from subscribing you to a Redux [store](#), inject `dispatch` into your component's props.

The fixed code looks like this:

AddTodo.js


```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { addTodo } from './TodoActions'

class AddTodo extends Component {
  handleClick() {
    // Works!
    this.props.dispatch(addTodo('Fix the issue'))
  }

  render() {
    return <button onClick={() => this.handleClick()}>Add</button>
  }
}

// In addition to the state, `connect` puts `dispatch` in our props.
export default connect()(AddTodo)
```

You can then pass `dispatch` down to other components manually, if you want to.

Make sure mapStateToProps is correct

It's possible you're correctly dispatching an [action](#) and applying your [reducer](#) but the corresponding [state](#) is not being correctly translated into props.

Something else doesn't work

Ask around on the [#redux Reactiflux](#) Discord channel, or [create an issue](#).

If you figure it out, [edit this document](#) as a courtesy to the next person having the same problem.

Glossary

This is a glossary of the core terms in Redux, along with their type signatures. The types are documented using [Flow notation](#).

State

```
type State = any
```

[State](#) (also called the [state tree](#)) is a broad term, but in the Redux API it usually refers to the single [state](#) value that is managed by the [store](#) and returned by `getState()`. It represents the entire [state](#) of a Redux application, which is often a deeply nested object.

By convention, the top-level [state](#) is an object or some other key-value collection like a Map, but technically it can be any type. Still, you should do your best to keep the [state](#) serializable. Don't put anything inside it that you can't easily turn into JSON.

Action

```
type Action = Object
```

An [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.

Actions must have a `type` field that indicates the type of [action](#) being performed. Types can be defined as constants and imported from another module. It's better to use strings for `type` than [Symbols](#) because strings are serializable.

Other than `type`, the structure of an [action](#) object is really up to you. If you're interested, check out [Flux Standard Action](#) for recommendations on how actions should be constructed.

See also [async action](#) below.

Reducer

```
type Reducer<S, A> = (state: S, action: A) => S
```

A [reducer](#) (also called a *reducing function*) 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.

Reducers are not unique to Redux—they are a fundamental concept in functional programming. Even most non-functional languages, like JavaScript, have a built-in API for reducing. In JavaScript, it's `Array.prototype.reduce()`.

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.

Reducers are the most important concept in Redux.

Do not put API calls into reducers.

Dispatching Function

```
type BaseDispatch = (a: Action) => Action
type Dispatch = (a: Action | AsyncAction) => any
```

A [dispatching function](#) (or simply *dispatch function*) is a function that accepts an [action](#) or an [async action](#); it then may or may not dispatch one or more actions to the [store](#).

We must distinguish between dispatching functions in general and the base `dispatch` function provided by the [store](#) instance without any [middleware](#).

The base dispatch function *always* 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](#).

[Middleware](#) wraps the base dispatch function. It allows the dispatch function to handle [async actions](#) in addition to actions. [Middleware](#) may transform, delay, ignore, or otherwise interpret actions or async actions before passing them to the next [middleware](#). See below for more information.

Action Creator

```
type ActionCreator = (...args: any) => Action | AsyncAction
```

An [action creator](#) is, quite simply, a function that creates an [action](#). Do not confuse the two terms—again, an [action](#) is a payload of information, and an [action creator](#) is a factory that creates an [action](#).

Calling an [action creator](#) only produces an [action](#), but does not dispatch it. You need to call the [store's](#) `dispatch` function to actually cause the mutation. Sometimes we say *bound action creators* to mean functions that call an [action creator](#) and immediately dispatch its result to a specific [store](#) instance.

If an [action creator](#) needs to read the current [state](#), perform an API call, or cause a side effect, like a routing transition, it should return an [async action](#) instead of an [action](#).

Async Action

```
type AsyncAction = any
```

An *async action* is a value that is sent to a [dispatching function](#), but is not yet ready for consumption by the [reducer](#). It will be transformed by [middleware](#) into an [action](#) (or a series of actions) before being sent to the base [dispatch\(\)](#) function. Async actions may have different types, depending on the [middleware](#) you use. They are often asynchronous primitives, like a Promise or a thunk, which are not passed to the [reducer](#) immediately, but trigger [action](#) dispatches once an operation has completed.

Middleware

```
type MiddlewareAPI = { dispatch: Dispatch, getState: () => State }  
type Middleware = (api: MiddlewareAPI) => (next: Dispatch) => Dispatch
```

A [middleware](#) is a higher-order function that composes a [dispatch function](#) to return a new dispatch function. It often turns [async actions](#) into actions.

[Middleware](#) is composable using function composition. It is useful for logging actions, performing side effects like routing, or turning an asynchronous API call into a series of synchronous actions.

See [applyMiddleware\(...middlewares\)](#) for a detailed look at [middleware](#).

Store

```
type Store = {  
  dispatch: Dispatch  
  getState: () => State  
  subscribe: (listener: () => void) => () => void  
  replaceReducer: (reducer: Reducer) => void  
}
```

A [store](#) is an object that holds the application's [state](#) tree.

There should only be a single [store](#) in a Redux app, as the composition happens on the [reducer](#) level.

- [dispatch\(action\)](#) is the base dispatch function described above.
- [getState\(\)](#) returns the current [state](#) of the [store](#).
- [subscribe\(listener\)](#) registers a function to be called on [state](#) changes.
- [replaceReducer\(nextReducer\)](#) can be used to implement hot reloading and code splitting. Most likely you won't use it.

See the complete [store API reference](#) for more details.

Store creator

```
type StoreCreator = (reducer: Reducer, preloadedState: ?State) => Store
```

A [store](#) creator is a function that creates a Redux [store](#). Like with [dispatching function](#), we must distinguish the base [store](#) creator, [createStore\(reducer, preloadedState\)](#) exported from the Redux package, from [store](#) creators that are returned from the [store](#) enhancers.

Store enhancer

```
type StoreEnhancer = (next: StoreCreator) => StoreCreator
```

A [store](#) enhancer is a higher-order function that composes a [store](#) creator to return a new, enhanced [store](#) creator. This is similar to [middleware](#) in that it allows you to alter the [store](#) interface in a composable way.

[Store](#) enhancers are much the same concept as higher-order components in React, which are also occasionally called “component enhancers”.

Because a [store](#) is not an instance, but rather a plain-object collection of functions, copies can be easily created and modified without mutating the original [store](#). There is an example in [compose](#) documentation demonstrating that.

Most likely you'll never write a [store](#) enhancer, but you may use the one provided by the [developer tools](#). It is what makes time travel possible without the app being aware it is happening. Amusingly, the [Redux middleware implementation](#) is itself a [store](#) enhancer.

API Reference

The Redux API surface is tiny. Redux defines a set of contracts for you to implement (such as [reducers](#)) and provides a few helper functions to tie these contracts together.

This section documents the complete Redux API. Keep in mind that Redux is only concerned with managing the [state](#). In a real app, you'll also want to use UI bindings like [react-redux](#).

Top-Level Exports

- [createStore\(reducer, \[preloadedState\], \[enhancer\]\)](#)
- [combineReducers\(reducers\)](#)
- [applyMiddleware\(...middlewares\)](#)
- [bindActionCreators\(actionCreators, dispatch\)](#)
- [compose\(...functions\)](#)

Store API

- [Store](#)
 - [getState\(\)](#)
 - [dispatch\(action\)](#)
 - [subscribe\(listener\)](#)
 - [replaceReducer\(nextReducer\)](#)

Importing

Every function described above is a top-level export. You can import any of them like this:

ES6

```
import { createStore } from 'redux'
```

ES5 (CommonJS)

```
var createStore = require('redux').createStore
```

ES5 (UMD build)

```
var createStore = Redux.createStore
```

createStore(reducer, [preloadedState], [enhancer])

Creates a Redux [store](#) that holds the complete [state](#) tree of your app.

There should only be a single [store](#) in your app.

Arguments

1. `reducer` (*Function*): A [reducing function](#) that returns the next [state tree](#), given the current [state](#) tree and an [action](#) to handle.
2. `[preloadedState]` (*any*): The initial [state](#). You may optionally specify it to hydrate the [state](#) from the server in universal apps, or to restore a previously serialized user session. If you produced `reducer` with `combineReducers`, this must be a plain object with the same shape as the keys passed to it. Otherwise, you are free to pass anything that your `reducer` can understand.
3. `[enhancer]` (*Function*): The [store](#) enhancer. You may optionally specify it to enhance the [store](#) with third-party capabilities such as [middleware](#), time travel, persistence, etc. The only [store](#) enhancer that ships with Redux is `applyMiddleware()`.

Returns

([Store](#)): An object that holds the complete [state](#) of your app. The only way to change its [state](#) is by [dispatching actions](#). You may also [subscribe](#) to the changes to its [state](#) to update the UI.

Example

```
import { createStore } from 'redux'

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.text])
    default:
      return state
  }
}

const store = createStore(todos, ['Use Redux'])

store.dispatch({
  type: 'ADD_TODO',
  text: 'Read the docs'
})

console.log(store.getState())
// [ 'Use Redux', 'Read the docs' ]
```

Tips

- Don't create more than one [store](#) in an application! Instead, use `combineReducers` to create a single root [reducer](#) out of many.

- It is up to you to choose the [state](#) format. You can use plain objects or something like [Immutable](#). If you're not sure, start with plain objects.
- If your [state](#) is a plain object, make sure you never mutate it! For example, instead of returning something like `Object.assign(state, newData)` from your reducers, return `Object.assign({}, state, newData)`. This way you don't override the previous `state`. You can also write `return { ...state, ...newData }` if you enable the [object spread operator proposal](#).
- For universal apps that run on the server, create a [store](#) instance with every request so that they are isolated. Dispatch a few data fetching actions to a [store](#) instance and wait for them to complete before rendering the app on the server.
- When a [store](#) is created, Redux dispatches a dummy [action](#) to your [reducer](#) to populate the [store](#) with the initial [state](#). You are not meant to handle the dummy [action](#) directly. Just remember that your [reducer](#) should return some kind of initial [state](#) if the [state](#) given to it as the first argument is `undefined`, and you're all set.
- To apply multiple [store](#) enhancers, you may use `compose()`.

Store

A **store** holds the whole **state tree** of your application. The only way to change the **state** inside it is to dispatch an **action** on it.

A **store** is not a class. It's just an object with a few methods on it. To create it, pass your root **reducing function** to `createStore` .

A Note for Flux Users

If you're coming from Flux, there is a single important difference you need to understand. Redux doesn't have a Dispatcher or support many stores. **Instead, there is just a single store with a single root reducing function.** As your app grows, instead of adding stores, you split the root **reducer** into smaller reducers independently operating on the different parts of the **state** tree. You can use a helper like `combineReducers` to combine them. This is similar to how there is just one root component in a React app, but it is composed out of many small components.

Store Methods

- `getState()`
- `dispatch(action)`
- `subscribe(listener)`
- `replaceReducer(nextReducer)`

Store Methods

getState()

Returns the current **state** tree of your application. It is equal to the last value returned by the **store's reducer**.

Returns

(any): The current **state** tree of your application.

dispatch(action)

Dispatches an **action**. This is the only way to trigger a **state** change.

The **store's** reducing function will be called with the current `getState()` result and the given `action` synchronously. Its return value will be considered the next **state**. It will be returned from `getState()` from now on, and the change listeners will immediately be notified.

A Note for Flux Users

If you attempt to call `dispatch` from inside the **reducer**, it will throw with an error saying “Reducers may not dispatch actions.” This is similar to “Cannot dispatch in a middle of dispatch” error in Flux, but doesn't cause the problems associated with it. In Flux, a dispatch is forbidden while Stores are handling the **action**

and emitting updates. This is unfortunate because it makes it impossible to dispatch actions from component lifecycle hooks or other benign places.

In Redux, subscriptions are called after the root [reducer](#) has returned the new [state](#), so you *may* dispatch in the subscription listeners. You are only disallowed to dispatch inside the reducers because they must have no side effects. If you want to cause a side effect in response to an [action](#), the right place to do this is in the potentially async [action creator](#).

Arguments

1. `action` (*Object*[†]): A plain object describing the change that makes sense for your application. Actions are the only way to get data into the [store](#), so any data, whether from the UI events, network callbacks, or other sources such as WebSockets needs to eventually be dispatched as actions. Actions must have a `type` field that indicates the type of [action](#) being performed. Types can be defined as constants and imported from another module. It's better to use strings for `type` than [Symbols](#) because strings are serializable. Other than `type`, the structure of an [action](#) object is really up to you. If you're interested, check out [Flux Standard Action](#) for recommendations on how actions could be constructed.

Returns

(*Object*[†]): The dispatched [action](#) (see notes).

Notes

[†] The “vanilla” [store](#) implementation you get by calling `createStore` only supports plain object actions and hands them immediately to the [reducer](#).

However, if you wrap `createStore` with `applyMiddleware`, the [middleware](#) can interpret actions differently, and provide support for dispatching [async actions](#). Async actions are usually asynchronous primitives like Promises, Observables, or thunks.

[Middleware](#) is created by the community and does not ship with Redux by default. You need to explicitly install packages like [redux-thunk](#) or [redux-promise](#) to use it. You may also create your own [middleware](#).

To learn how to describe asynchronous API calls, read the current [state](#) inside [action](#) creators, perform side effects, or chain them to execute in a sequence, see the examples for `applyMiddleware`.

Example

```
import { createStore } from 'redux'
const store = createStore(todos, ['Use Redux'])

function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

store.dispatch(addTodo('Read the docs'))
store.dispatch(addTodo('Read about the middleware'))
```

subscribe(listener)

Adds a change listener. It will be called any time an [action](#) is dispatched, and some part of the [state](#) tree may potentially have changed. You may then call [getState\(\)](#) to read the current [state](#) tree inside the callback.

You may call [dispatch\(\)](#) from a change listener, with the following caveats:

1. The listener should only call [dispatch\(\)](#) either in response to user actions or under specific conditions (e. g. dispatching an [action](#) when the [store](#) has a specific field). Calling [dispatch\(\)](#) without any conditions is technically possible, however it leads to an infinite loop as every [dispatch\(\)](#) call usually triggers the listener again.
2. The subscriptions are snapshotted just before every [dispatch\(\)](#) call. If you subscribe or unsubscribe while the listeners are being invoked, this will not have any effect on the [dispatch\(\)](#) that is currently in progress. However, the next [dispatch\(\)](#) call, whether nested or not, will use a more recent snapshot of the subscription list.
3. The listener should not expect to see all [state](#) changes, as the [state](#) might have been updated multiple times during a nested [dispatch\(\)](#) before the listener is called. It is, however, guaranteed that all subscribers registered before the [dispatch\(\)](#) started will be called with the latest [state](#) by the time it exits.

It is a low-level API. Most likely, instead of using it directly, you'll use React (or other) bindings. If you commonly use the callback as a hook to react to [state](#) changes, you might want to [write a custom observeStore utility](#). The [Store](#) is also an [Observable](#), so you can [subscribe](#) to changes with libraries like [RxJS](#).

To unsubscribe the change listener, invoke the function returned by [subscribe](#).

Arguments

1. [listener](#) (*Function*): The callback to be invoked any time an [action](#) has been dispatched, and the [state](#) tree might have changed. You may call [getState\(\)](#) inside this callback to read the current [state](#) tree. It is reasonable to expect that the [store's reducer](#) is a pure function, so you may compare references to some deep path in the [state](#) tree to learn whether its value has changed.

Returns

(*Function*): A function that unsubscribes the change listener.

Example

```
function select(state) {
  return state.some.deep.property
}

let currentValue
function handleChange() {
  let previousValue = currentValue
  currentValue = select(store.getState())

  if (previousValue !== currentValue) {
    console.log(
      'Some deep nested property changed from',
      previousValue,
      'to',
      currentValue
    )
  }
}
```

```
const unsubscribe = store.subscribe(handleChange)
unsubscribe()
```

replaceReducer(nextReducer)

Replaces the [reducer](#) currently used by the [store](#) to calculate the [state](#).

It is an advanced API. You might need this if your app implements code splitting, and you want to load some of the reducers dynamically. You might also need this if you implement a hot reloading mechanism for Redux.

Arguments

1. `nextReducer` (*Function*) The next [reducer](#) for the [store](#) to use.

combineReducers(reducers)

As your app grows more complex, you'll want to split your [reducing function](#) into separate functions, each managing independent parts of the [state](#).

The `combineReducers` helper function turns an object whose values are different reducing functions into a single reducing function you can pass to `createStore`.

The resulting [reducer](#) calls every child [reducer](#), and gathers their results into a single [state](#) object. **The [state](#) produced by `combineReducers()` namespaces the states of each [reducer](#) under their keys as passed to `combineReducers()`**

Example:

```
rootReducer = combineReducers({potato: potatoReducer, tomato: tomatoReducer})
// This would produce the following state object
{
  potato: {
    // ... potatoes, and other state managed by the potatoReducer ...
  },
  tomato: {
    // ... tomatoes, and other state managed by the tomatoReducer, maybe some nice sauce? ...
  }
}
```

You can control [state](#) key names by using different keys for the reducers in the passed object. For example, you may call `combineReducers({ todos: myTodosReducer, counter: myCounterReducer })` for the [state](#) shape to be `{ todos, counter }`.

A popular convention is to name reducers after the [state](#) slices they manage, so you can use ES6 property shorthand notation: `combineReducers({ counter, todos })`. This is equivalent to writing `combineReducers({ counter: counter, todos: todos })`.

A Note for Flux Users

This function helps you organize your reducers to manage their own slices of [state](#), similar to how you would have different Flux Stores to manage different [state](#). With Redux, there is just one [store](#), but

`combineReducers` helps you keep the same logical division between reducers.

Arguments

1. `reducers` (*Object*): An object whose values correspond to different reducing functions that need to be combined into one. See the notes below for some rules every passed [reducer](#) must follow.

Earlier documentation suggested the use of the ES6 `import * as reducers` syntax to obtain the reducers object. This was the source of a lot of confusion, which is why we now recommend exporting a single [reducer](#) obtained using `combineReducers()` from `reducers/index.js` instead. An example is included below.

Returns

(*Function*): A [reducer](#) that invokes every [reducer](#) inside the `reducers` object, and constructs a [state](#) object with the same shape.

Notes

This function is mildly opinionated and is skewed towards helping beginners avoid common pitfalls. This is why it attempts to enforce some rules that you don't have to follow if you write the root `reducer` manually.

Any `reducer` passed to `combineReducers` must satisfy these rules:

- For any `action` that is not recognized, it must return the `state` given to it as the first argument.
- It must never return `undefined`. It is too easy to do this by mistake via an early `return` statement, so `combineReducers` throws if you do that instead of letting the error manifest itself somewhere else.
- If the `state` given to it is `undefined`, it must return the initial `state` for this specific `reducer`. According to the previous rule, the initial `state` must not be `undefined` either. It is handy to specify it with ES6 optional arguments syntax, but you can also explicitly check the first argument for being `undefined`.

While `combineReducers` attempts to check that your reducers conform to some of these rules, you should remember them, and do your best to follow them. `combineReducers` will check your reducers by passing `undefined` to them; this is done even if you specify initial `state` to `Redux.createStore(combineReducers(...), initialState)`.

Therefore, you **must** ensure your reducers work properly when receiving `undefined` as `state`, even if you never intend for them to actually receive `undefined` in your own code.

Example

reducers/todos.js

```
export default function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([action.text])
    default:
      return state
  }
}
```

reducers/counter.js

```
export default function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    default:
      return state
  }
}
```

reducers/index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import counter from './counter'

export default combineReducers({
  todos,
```

```
    counter
  })
```

App.js

```
import { createStore } from 'redux'
import reducer from './reducers/index'

const store = createStore(reducer)
console.log(store.getState())
// {
//   counter: 0,
//   todos: []
// }

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})
console.log(store.getState())
// {
//   counter: 0,
//   todos: [ 'Use Redux' ]
// }
```

Tips

- This helper is just a convenience! You can write your own `combineReducers` that [works differently](#), or even assemble the [state](#) object from the child reducers manually and write a root reducing function explicitly, like you would write any other function.
- You may call `combineReducers` at any level of the [reducer](#) hierarchy. It doesn't have to happen at the top. In fact you may use it again to split the child reducers that get too complicated into independent grandchildren, and so on.

applyMiddleware(...middleware)

Middleware is the suggested way to extend Redux with custom functionality. **Middleware** lets you wrap the **store's** **dispatch** method for fun and profit. The key feature of **middleware** is that it is composable. Multiple **middleware** can be combined together, where each **middleware** requires no knowledge of what comes before or after it in the chain.

The most common use case for **middleware** is to support asynchronous actions without much boilerplate code or a dependency on a library like **Rx**. It does so by letting you dispatch **async actions** in addition to normal actions.

For example, **redux-thunk** lets the **action** creators invert control by dispatching functions. They would receive **dispatch** as an argument and may call it asynchronously. Such functions are called *thunks*. Another example of **middleware** is **redux-promise**. It lets you dispatch a **Promise** **async action**, and dispatches a normal **action** when the Promise resolves.

Middleware is not baked into **createStore** and is not a fundamental part of the Redux architecture, but we consider it useful enough to be supported right in the core. This way, there is a single standard way to extend **dispatch** in the ecosystem, and different **middleware** may compete in expressiveness and utility.

Arguments

- ...middleware** (*arguments*): Functions that conform to the Redux **middleware API**. Each **middleware** receives **Store's** **dispatch** and **getState** functions as named arguments, and returns a function. That function will be given the **next** **middleware's** **dispatch** method, and is expected to return a function of **action** calling **next(action)** with a potentially different argument, or at a different time, or maybe not calling it at all. The last **middleware** in the chain will receive the real **store's** **dispatch** method as the **next** parameter, thus ending the chain. So, the **middleware** signature is `({ getState, dispatch }) => next => action`.

Returns

(Function) A **store** enhancer that applies the given **middleware**. The **store** enhancer signature is `createStore => createStore` but the easiest way to apply it is to pass it to `createStore()` as the last **enhancer** argument.

Example: Custom Logger Middleware

```
import { createStore, applyMiddleware } from 'redux'
import todos from './reducers'

function logger({ getState }) {
  return next => action => {
    console.log('will dispatch', action)

    // Call the next dispatch method in the middleware chain.
    const returnValue = next(action)

    console.log('state after dispatch', getState())

    // This will likely be the action itself, unless
    // a middleware further in chain changed it.
    return returnValue
  }
}
```



```
const store = createStore(todos, ['Use Redux'], applyMiddleware(logger))

store.dispatch({
  type: 'ADD_TODO',
  text: 'Understand the middleware'
})
// (These lines will be logged by the middleware:)
// will dispatch: { type: 'ADD_TODO', text: 'Understand the middleware' }
// state after dispatch: [ 'Use Redux', 'Understand the middleware' ]
```

Example: Using Thunk Middleware for Async Actions

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import * as reducers from './reducers'

const reducer = combineReducers(reducers)
// applyMiddleware supercharges createStore with middleware:
const store = createStore(reducer, applyMiddleware(thunk))

function fetchSecretSauce() {
  return fetch('https://www.google.com/search?q=secret+sauce')
}

// These are the normal action creators you have seen so far.
// The actions they return can be dispatched without any middleware.
// However, they only express “facts” and not the “async flow”.
function makeASandwich(forPerson, secretSauce) {
  return {
    type: 'MAKE_SANDWICH',
    forPerson,
    secretSauce
  }
}

function apologize(fromPerson, toPerson, error) {
  return {
    type: 'APOLOGIZE',
    fromPerson,
    toPerson,
    error
  }
}

function withdrawMoney(amount) {
  return {
    type: 'WITHDRAW',
    amount
  }
}

// Even without middleware, you can dispatch an action:
store.dispatch(withdrawMoney(100))

// But what do you do when you need to start an asynchronous action,
// such as an API call, or a router transition?

// Meet thunks.
// A thunk is a function that returns a function.
// This is a thunk.
function makeASandwichWithSecretSauce(forPerson) {
  // Invert control!
  // Return a function that accepts `dispatch` so we can dispatch later.
  // Thunk middleware knows how to turn thunk async actions into actions.
```

```

    return function(dispatch) {
      return fetchSecretSauce().then(
        sauce => dispatch(makeASandwich(forPerson, sauce)),
        error => dispatch(apologize('The Sandwich Shop', forPerson, error))
      )
    }
  }
}

// Thunk middleware lets me dispatch thunk async actions
// as if they were actions!
store.dispatch(makeASandwichWithSecretSauce('Me'))

// It even takes care to return the thunk's return value
// from the dispatch, so I can chain Promises as long as I return them.
store.dispatch(makeASandwichWithSecretSauce('My wife')).then(() => {
  console.log('Done!')
})

// In fact I can write action creators that dispatch
// actions and async actions from other action creators,
// and I can build my control flow with Promises.
function makeSandwichesForEverybody() {
  return function(dispatch, getState) {
    if (!getState().sandwiches.isShopOpen) {
      // You don't have to return Promises, but it's a handy convention
      // so the caller can always call .then() on async dispatch result.
      return Promise.resolve()
    }

    // We can dispatch both plain object actions and other thunks,
    // which lets us compose the asynchronous actions in a single flow.
    return dispatch(makeASandwichWithSecretSauce('My Grandma'))
      .then(() =>
        Promise.all([
          dispatch(makeASandwichWithSecretSauce('Me')),
          dispatch(makeASandwichWithSecretSauce('My wife'))
        ])
      )
      .then(() => dispatch(makeASandwichWithSecretSauce('Our kids')))
      .then(() =>
        dispatch(
          getState().myMoney > 42
            ? withdrawMoney(42)
            : apologize('Me', 'The Sandwich Shop')
        )
      )
  }
}

// This is very useful for server side rendering, because I can wait
// until data is available, then synchronously render the app.

import { renderToString } from 'react-dom/server'

store
  .dispatch(makeSandwichesForEverybody())
  .then(() => response.send(renderToString(<MyApp store={store} />)))

// I can also dispatch a thunk async action from a component
// any time its props change to load the missing data.

import { connect } from 'react-redux'
import { Component } from 'react'

class SandwichShop extends Component {
  componentDidMount() {

```

```

    this.props.dispatch(makeASandwichWithSecretSauce(this.props.forPerson))
  }

  componentDidUpdate(prevProps) {
    if (prevProps.forPerson !== this.props.forPerson) {
      this.props.dispatch(makeASandwichWithSecretSauce(this.props.forPerson))
    }
  }

  render() {
    return <p>{this.props.sandwiches.join('mustard')}</p>
  }
}

export default connect(state => ({
  sandwiches: state.sandwiches
}))(SandwichShop)

```

Tips

- **Middleware** only wraps the **store's** **dispatch** function. Technically, anything a **middleware** can do, you can do manually by wrapping every **dispatch** call, but it's easier to manage this in a single place and define **action** transformations on the scale of the whole project.
- If you use other **store** enhancers in addition to **applyMiddleware**, make sure to put **applyMiddleware** before them in the composition chain because the **middleware** is potentially asynchronous. For example, it should go before **redux-devtools** because otherwise the DevTools won't see the raw actions emitted by the Promise **middleware** and such.
- If you want to conditionally apply a **middleware**, make sure to only import it when it's needed:

```

let middleware = [a, b]
if (process.env.NODE_ENV !== 'production') {
  const c = require('some-debug-middleware')
  const d = require('another-debug-middleware')
  middleware = [...middleware, c, d]
}

const store = createStore(
  reducer,
  preloadedState,
  applyMiddleware(...middleware)
)

```

This makes it easier for bundling tools to cut out unneeded modules and reduces the size of your builds.

- Ever wondered what **applyMiddleware** itself is? It ought to be an extension mechanism more powerful than the **middleware** itself. Indeed, **applyMiddleware** is an example of the most powerful Redux extension mechanism called **store enhancers**. It is highly unlikely you'll ever want to write a **store** enhancer yourself. Another example of a **store** enhancer is **redux-devtools**. **Middleware** is less powerful than a **store** enhancer, but it is easier to write.
- **Middleware** sounds much more complicated than it really is. The only way to really understand **middleware** is to see how the existing **middleware** works, and try to write your own. The function nesting can be intimidating, but most of the **middleware** you'll find are, in fact, 10-liners, and the nesting and composability is what makes the **middleware** system powerful.
- To apply multiple **store** enhancers, you may use **compose()**.

bindActionCreators(actionCreators, dispatch)

Turns an object whose values are [action creators](#), into an object with the same keys, but with every [action](#) creator wrapped into a `dispatch` call so they may be invoked directly.

Normally you should just call `dispatch` directly on your `Store` instance. If you use Redux with React, `react-redux` will provide you with the `dispatch` function so you can call it directly, too.

The only use case for `bindActionCreators` is when you want to pass some [action](#) creators down to a component that isn't aware of Redux, and you don't want to pass `dispatch` or the Redux `store` to it.

For convenience, you can also pass an [action](#) creator as the first argument, and get a dispatch wrapped function in return.

Parameters

1. `actionCreators` (*Function or Object*): An [action creator](#), or an object whose values are [action](#) creators.
2. `dispatch` (*Function*): A `dispatch` function available on the `Store` instance.

Returns

(*Function or Object*): An object mimicking the original object, but with each function immediately dispatching the [action](#) returned by the corresponding [action](#) creator. If you passed a function as `actionCreators`, the return value will also be a single function.

Example

TodoActionCreators.js

```
export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    text
  }
}

export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  }
}
```

SomeComponent.js

```
import { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

import * as TodoActionCreators from './TodoActionCreators'
console.log(TodoActionCreators)
// {
//   addTodo: Function,
```

```

//   removeTodo: Function
// }

class TodoListContainer extends Component {
  constructor(props) {
    super(props)

    const { dispatch } = props

    // Here's a good use case for bindActionCreators:
    // You want a child component to be completely unaware of Redux.
    // We create bound versions of these functions now so we can
    // pass them down to our child later.

    this.boundActionCreators = bindActionCreators(TodoActionCreators, dispatch)
    console.log(this.boundActionCreators)
    // {
    //   addTo: Function,
    //   removeTodo: Function
    // }
  }

  componentDidMount() {
    // Injected by react-redux:
    let { dispatch } = this.props

    // Note: this won't work:
    // TodoActionCreators.addTo('Use Redux')

    // You're just calling a function that creates an action.
    // You must dispatch the action, too!

    // This will work:
    let action = TodoActionCreators.addTo('Use Redux')
    dispatch(action)
  }

  render() {
    // Injected by react-redux:
    let { todos } = this.props

    return <TodoList todos={todos} {...this.boundActionCreators} />

    // An alternative to bindActionCreators is to pass
    // just the dispatch function down, but then your child component
    // needs to import action creators and know about them.

    // return <TodoList todos={todos} dispatch={dispatch} />
  }
}

export default connect(state => ({ todos: state.todos }))(TodoListContainer)

```

Tips

- You might ask: why don't we bind the [action](#) creators to the [store](#) instance right away, like in classical Flux? The problem is that this won't work well with universal apps that need to render on the server. Most likely you want to have a separate [store](#) instance per request so you can prepare them with different data, but binding [action](#) creators during their definition means you're stuck with a single [store](#) instance for all requests.

- If you use ES5, instead of `import * as` syntax you can just pass `require('./TodoActionCreators')` to `bindActionCreators` as the first argument. The only thing it cares about is that the values of the `actionCreators` properties are functions. The module system doesn't matter.

compose(...functions)

Composes functions from right to left.

This is a functional programming utility, and is included in Redux as a convenience.

You might want to use it to apply several [store enhancers](#) in a row.

Arguments

1. (*arguments*): The functions to compose. Each function is expected to accept a single parameter. Its return value will be provided as an argument to the function standing to the left, and so on. The exception is the right-most argument which can accept multiple parameters, as it will provide the signature for the resulting composed function.

Returns

(*Function*): The final function obtained by composing the given functions from right to left.

Example

This example demonstrates how to use `compose` to enhance a [store](#) with `applyMiddleware` and a few developer tools from the [redux-devtools](#) package.

```
import { createStore, applyMiddleware, compose } from 'redux'
import thunk from 'redux-thunk'
import DevTools from './containers/DevTools'
import reducer from './reducers'

const store = createStore(
  reducer,
  compose(
    applyMiddleware(thunk),
    DevTools.instrument()
  )
)
```

Tips

- All `compose` does is let you write deeply nested function transformations without the rightward drift of the code. Don't give it too much credit!

Feedback

We appreciate feedback from the community. You can post feature requests and bug reports on [Product Pains](#).

A large, empty rectangular box with a thin black border, intended for users to provide feedback, feature requests, or bug reports.