
Table of Contents

Table of Contents

Read Me	1.1
Introduction	1.2
Motivation	1.2.1
Three Principles	1.2.2
Prior Art	1.2.3
Ecosystem	1.2.4
Examples	1.2.5
Basics	1.3
Actions	1.3.1
Reducers	1.3.2
Store	1.3.3
Data Flow	1.3.4
Usage with React	1.3.5
Example: Todo List	1.3.6
Advanced	1.4
Async Actions	1.4.1
Async Flow	1.4.2
Middleware	1.4.3
Usage with React Router	1.4.4
Example: Reddit API	1.4.5
Next Steps	1.4.6
Recipes	1.5
Migrating to Redux	1.5.1
Using Object Spread Operator	1.5.2
Reducing Boilerplate	1.5.3

Server Rendering	1.5.4
Writing Tests	1.5.5
Computing Derived Data	1.5.6
Implementing Undo History	1.5.7
FAQ	1.6
Troubleshooting	1.7
Glossary	1.8
API Reference	1.9
createStore	1.9.1
Store	1.9.2
combineReducers	1.9.3
applyMiddleware	1.9.4
bindActionCreators	1.9.5
compose	1.9.6
Change Log	1.10
Patrons	1.11
Feedback	1.12

Table of Contents

- [Read Me](#)
- [Introduction](#)
 - [Motivation](#)
 - [Three Principles](#)
 - [Prior Art](#)
 - [Ecosystem](#)
 - [Examples](#)
- [Basics](#)
 - [Actions](#)
 - [Reducers](#)
 - [Store](#)
 - [Data Flow](#)
 - [Usage with React](#)
 - [Example: Todo List](#)
- [Advanced](#)
 - [Async Actions](#)
 - [Async Flow](#)
 - [Middleware](#)
 - [Usage with React Router](#)
 - [Example: Reddit API](#)
 - [Next Steps](#)
- [Recipes](#)
 - [Migrating to Redux](#)
 - [Using Object Spread Operator](#)
 - [Reducing Boilerplate](#)
 - [Server Rendering](#)
 - [Writing Tests](#)
 - [Computing Derived Data](#)
 - [Implementing Undo History](#)
- [FAQ](#)
- [Troubleshooting](#)
- [Glossary](#)
- [API Reference](#)
 - [`createStore`](#)

- [Store](#)
- [combineReducers](#)
- [applyMiddleware](#)
- [bindActionCreators](#)
- [compose](#)
- [Change Log](#)
- [Patrons](#)
- [Feedback](#)

Introduction

- Motivation
- Three Principles
- Prior Art
- 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 the active route, the selected tab, whether to show a spinner or not, should pagination controls be displayed, 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 wasn'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.

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 introspect 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 mutate 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 mutate. Because all mutations 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'
let reducer = combineReducers({ visibilityFilter, todos })
let 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

Can Redux be considered a [Flux](#) implementation?

[Yes](#), and [no](#).

(Don't worry, [Flux creators approve of it](#), if that's all you wanted to know.)

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](#).

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.

Rx

[Reactive Extensions](#) (and their undergoing [modern rewrite](#)) are 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 Rx? Sure! They work great together. For example, it is easy to expose a Redux [store](#) as an observable:

```
function toObservable(store) {
  return {
    subscribe({ onNext }) {
      let dispose = store.subscribe(() => onNext(store.getState()))
      onNext(store.getState())
      return { dispose }
    }
  }
}
```

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!

Ecosystem

Redux is a tiny library, but its contracts and APIs are carefully chosen to spawn an ecosystem of tools and extensions.

For an extensive list of everything related to Redux, we recommend [Awesome Redux](#). It contains examples, boilerplates, [middleware](#), utility libraries, and more. [React/Redux Links](#) contains tutorials and other useful resources for anyone learning React or Redux, and [Redux Ecosystem Links](#) lists many Redux-related libraries and addons.

On this page we will only feature a few of them that the Redux maintainers have vetted personally. 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.

Learning Redux

Screencasts

- [Getting Started with Redux](#) — Learn the basics of Redux directly from its creator (30 free videos)

Example Apps

- [Official Examples](#) — A few official examples covering different Redux techniques
- [SoundRedux](#) — A SoundCloud client built with Redux
- [grafgiti](#) — Create graffiti on your GitHub contributions wall

Tutorials and Articles

- [Redux Tutorial](#) — Learn how to use Redux step by step
- [Redux Egghead Course Notes](#) — Notes on the Redux [Egghead video course](#)
- [Integrating Data with React Native](#) — An intro to using Redux with React Native
- [What the Flux?! Let's Redux.](#) — An intro to Redux
- [Leveling Up with React: Redux](#) — Another great intro to Redux
- [A cartoon intro to Redux](#) — A visual explanation of Redux data flow

- [Understanding Redux](#) — Learn the basic concepts of Redux
- [Handcrafting an Isomorphic Redux Application \(With Love\)](#) — A guide to creating a universal app with data fetching and routing
- [Full-Stack Redux Tutorial](#) — A comprehensive guide to test-first development with Redux, React, and Immutable
- [Getting Started with React, Redux, and Immutable](#) — A test-driven intro to refactoring a React app to use Redux
- [Secure Your React and Redux App with JWT Authentication](#) — Learn how to add JWT authentication to your React and Redux app
- [Understanding Redux Middleware](#) — In-depth guide to implementing Redux middleware
- [Angular 2 — Introduction to Redux](#) — An introduction to Redux fundamental concepts with an example in Angular 2
- [Working with VK API \(in Russian\)](#) — A tutorial in Russian that demonstrates creating an app that consumes VK API

Talks

- [Live React: Hot Reloading and Time Travel](#) — See how constraints enforced by Redux make hot reloading with time travel easy
- [Cleaning the Tar: Using React within the Firefox Developer Tools](#) — Learn how to gradually migrate existing MVC applications to Redux
- [Redux: Simplifying Application State](#) — An intro to Redux architecture

Using Redux

Bindings

- [react-redux](#) — React
- [ng-redux](#) — Angular
- [ng2-redux](#) — Angular 2
- [backbone-redux](#) — Backbone
- [redux-falcor](#) — Falcor
- [deku-redux](#) — Deku

Middleware

- [redux-thunk](#) — The easiest way to write async `action` creators

- [redux-promise](#) — FSA-compliant promise [middleware](#)
- [redux-axios-middleware](#) — Redux [middleware](#) for fetching data with axios HTTP client
- [redux-observable](#) — RxJS [middleware](#) for Redux
- [redux-rx](#) — RxJS utilities for Redux, including a [middleware](#) for Observable
- [redux-logger](#) — Log every Redux [action](#) and the next [state](#)
- [redux-immutable-state-invariant](#) — Warns about [state](#) mutations in development
- [redux-unhandled-action](#) — Warns about actions that produced no [state](#) changes in development
- [redux-analytics](#) — Analytics [middleware](#) for Redux
- [redux-gen](#) — Generator [middleware](#) for Redux
- [redux-saga](#) — An alternative side effect model for Redux apps
- [redux-action-tree](#) — Composable Cerebral-style signals for Redux

Routing

- [react-router-redux](#) — Ruthlessly simple bindings to keep React Router and Redux in sync
- [redial](#) — Universal data fetching and route lifecycle management for React that works great with Redux

Components

- [redux-form](#) — Keep React form [state](#) in Redux
- [react-redux-form](#) — Create forms easily in React with Redux

Enhancers

- [redux-batched-subscribe](#) — Customize batching and debouncing calls to the [store](#) subscribers
- [redux-history-transitions](#) — History transitions based on arbitrary actions
- [redux-optimist](#) — Optimistically apply actions that can be later committed or reverted
- [redux-optimistic-ui](#) — A [reducer](#) enhancer to enable type-agnostic optimistic updates
- [redux-undo](#) — Effortless undo/redo and [action](#) history for your reducers
- [redux-ignore](#) — Ignore redux actions by array or filter function
- [redux-recycle](#) — Reset the redux [state](#) on certain actions

- [redux-batched-actions](#) — Dispatch several actions with a single subscriber notification
- [redux-search](#) — Automatically index resources in a web worker and search them without blocking
- [redux-electron-store](#) — [Store](#) enhancers that synchronize Redux stores across Electron processes
- [redux-loop](#) — Sequence effects purely and naturally by returning them from your reducers
- [redux-side-effects](#) — Utilize Generators for declarative yielding of side effects from your pure reducers

Utilities

- [reselect](#) — Efficient derived data selectors inspired by NuclearJS
- [normalizr](#) — Normalize nested API responses for easier consumption by the reducers
- [redux-actions](#) — Reduces the boilerplate in writing reducers and [action](#) creators
- [redux-act](#) — An opinionated library for making reducers and [action](#) creators
- [redux-transducers](#) — Transducer utilities for Redux
- [redux-immutable](#) — Used to create an equivalent function of Redux `combineReducers` that works with [Immutable.js state](#).
- [redux-tcomb](#) — Immutable and type-checked [state](#) and actions for Redux
- [redux-mock-store](#) — Mock redux [store](#) for testing your app
- [redux-actions-assertions](#) — Assertions for Redux actions testing

DevTools

- [Redux DevTools](#) — An [action](#) logger with time travel UI, hot reloading and error handling for the reducers, [first demoed at React Europe](#)
- [Redux DevTools Extension](#) — A Chrome extension wrapping Redux DevTools and providing additional functionality

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

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

Translations

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

More

[Awesome Redux](#) is an extensive list of Redux-related repositories.

[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.

Examples

Redux is distributed with a few examples in its [source code](#).

Counter Vanilla

Run the [Counter Vanilla](#) example:

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

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/reactjs/redux.git  
  
cd redux/examples/counter  
npm install  
npm start  
  
open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/todos
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/todos-with-undo
npm install
npm start

open http://localhost:3000/
```

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.

TodoMVC

Run the [TodoMVC](#) example:

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

cd redux/examples/todomvc
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/shopping-cart
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/tree-view
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/async
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

cd redux/examples/universal
npm install
npm start

open http://localhost:3000/
```

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/reactjs/redux.git

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

open http://localhost:3000/
```

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 [Awesome Redux](#).

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

[Actions](#) describe the fact that *something happened*, but don't specify how the application's [state](#) changes in response. This is the job of a [reducer](#).

Designing the State Shape

In Redux, all 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:

```
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, but is not implemented by most browsers yet. You'll 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! Let's extend our `reducer` to handle `ADD_TODO`.

```

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 `react-`

[addons-update](#), [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:
    case TOGGLE_TODO:
      return Object.assign({}, state, {
        todos: todos(state.todos, action)
      })
    default:
      return state
  }
}

```

Note that `todos` also accepts `state` —but it's an array! Now `todoApp` just gives it the 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:

```
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` anymore. 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 completely 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 completely 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 combining their results into a single object again. [It's not magic.](#)

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 don't use it anymore 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'
let 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.

```
let 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
let 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 functions. Call them, and make assertions on what they return.

Source Code

index.js

```

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

let 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.

2. **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.

3. 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`!

4. 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 frameworks 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.

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 [npmdcn](#) (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 embrace the idea of **separating presentational and container components**. If you're not familiar with these terms, [read about them first](#), and then come back. They are important, so we'll wait!

Finished reading the article? Let's recount their differences:

	Presentational Components	Container Components
Purpose	How things look (markup, styles)	How things work (data fetching, <code>state</code> updates)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux <code>state</code>
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`.

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.

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 todo should appear crossed out.
 - `onClick()` is a callback to invoke when a todo is clicked.
- `Link` is a link with a callback.
 - `onClick()` is a callback to invoke when 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.

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.

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 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.

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, { PropTypes } from 'react'

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, { PropTypes } from 'react'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map(todo =>
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => onTodoClick(todo.id)}
      />
    )}
  </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, { PropTypes } from 'react'

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'

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

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

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 tells 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_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)
  }
}
```

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
```

Other Components

containers/AddTodo.js

```

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

```

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'

let 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!

Example: Todo List

This is the complete source code of the tiny todo app we built during the [basics tutorial](#).

Entry Point

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'

let store = createStore(todoApp)

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

Action Creators

actions/index.js

Example: Todo List

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

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

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

Reducers

reducers/todos.js

Example: Todo List

```
const todo = (state, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return {
        id: action.id,
        text: action.text,
        completed: false
      }
    case 'TOGGLE_TODO':
      if (state.id !== action.id) {
        return state
      }

      return Object.assign({}, state, {
        completed: !state.completed
      })
    default:
      return state
  }
}

const todos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        todo(undefined, action)
      ]
    case 'TOGGLE_TODO':
      return state.map(t =>
        todo(t, action)
      )
    default:
      return state
  }
}

export default todos
```

reducers/visibilityFilter.js

Example: Todo List

```
const visibilityFilter = (state = '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'

const todoApp = combineReducers({
  todos,
  visibilityFilter
})

export default todoApp
```

Presentational Components

components/Todo.js

Example: Todo List

```
import React, { PropTypes } from 'react'

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, { PropTypes } from 'react'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map(todo =>
      <Todo
        key={todo.id}
        {...todo}
        onClick={() => onTodoClick(todo.id)}
      />
    )}
  </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, { PropTypes } from 'react'

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

Example: Todo List

```
import React from 'react'
import FilterLink from '../containers/FilterLink'

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

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

Example: Todo List

```
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
```

containers/FilterLink.js

Example: Todo List

```
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
```

Other Components

`containers/AddTodo.js`

Example: Todo List

```
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
```

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 select 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 can 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

```
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'

export 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'

export 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 `fetchedPageCount` 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 terse 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

```
import fetch from 'isomorphic-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()
  }
}

// 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,
}
```

Async Actions

```
// 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(`http://www.reddit.com/r/${subreddit}.json`)
    .then(response => response.json())
    .then(json =>

      // We can dispatch many times!
      // Here, we update the app state with the results of the API call.

      dispatch(receivePosts(subreddit, json))
    )

  // In a real world app, you also want to
  // catch any error in the network call.
}

}
```

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 `isomorphic-fetch` library:

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

```
import fetch from 'isomorphic-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()
  }
}

function fetchPosts(subreddit) {
  return dispatch => {
    dispatch(requestPosts(subreddit))
    return fetch(`http://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
```

Async Actions

```
// 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 dispatch Observables with [redux-rx](#). 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 interpret anything you dispatch, and in turn, can pass actions to the next [middleware](#) in 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 you saw 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
  ① dispatching: Object {type: "ADD_TODO", text: "Use Redux"}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[1]}
▼ ADD_TODO
  ① dispatching: Object {type: "ADD_TODO", text: "Learn about middleware"}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}
▼ COMPLETE_TODO
  ① dispatching: Object {type: "COMPLETE_TODO", index: 0}
    next state: ► Object {visibilityFilter: "SHOW_ALL", todos: Array[2]}
▼ SET_VISIBILITY_FILTER
  ① dispatching: Object {type: "SET_VISIBILITY_FILTER", filter: "SHOW_COMPLETED"}
    next state: ► Object {visibilityFilter: "SHOW_COMPLETED", todos: Array[2]}

```

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:

```

let 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:

```
let 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) {
  let 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) {
  let 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) {
  let 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:
  let 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](#).
- 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.

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'

let todoApp = combineReducers(reducers)
let 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(action.type)  
  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)  
  }  
  
  let 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 => {
  let 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) {
    let 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.)
let todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  applyMiddleware(
    rafScheduler,
    timeoutScheduler,
    thunk,
    vanillaPromise,
```

```
  readyStatePromise,  
  logger,  
  crashReporter  
)  
)
```

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 'isomorphic-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,
```

Example: Reddit API

```
 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(`http://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)
      })
  }
}

```

Example: Reddit API

```
        })
      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(initialState) {
  return createStore(
    rootReducer,
    initialState,
    applyMiddleware(
      thunkMiddleware,
      loggerMiddleware
    )
  )
}
```

Container Components

containers/Root.js

Example: Reddit API

```
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, PropTypes } from 'react'
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))
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.selectedSubreddit !== this.props.selectedSubreddit) {
      const { dispatch, selectedSubreddit } = nextProps
      dispatch(fetchPostsIfNeeded(selectedSubreddit))
    }
  }

  handleChange(nextSubreddit) {
    this.props.dispatch(selectSubreddit(nextSubreddit))
  }

  handleRefreshClick(e) {
```

Example: Reddit API

```
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 &&
          <a href="#" onClick={this.handleRefreshClick}>
            Refresh
          </a>
        }
      </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
}
```

Example: Reddit API

```
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

Example: Reddit API

```
import React, { Component, PropTypes } from 'react'

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

Example: Reddit API

```
import React, { PropTypes, Component } from 'react'

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
}
```

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.

- [Migrating to Redux](#)
- [Using Object Spread Operator](#)
- [Reducing Boilerplate](#)
- [Server Rendering](#)
- [Writing Tests](#)
- [Computing Derived Data](#)
- [Implementing Undo History](#)

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.

It is also possible to do the reverse and migrate from Redux to any of these libraries following the same steps.

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.

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](#) proposed for the next versions of JavaScript which 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)
}))
```

Since the object spread syntax is still a Stage 2 proposal for ECMAScript you'll need to use a transpiler such as [Babel](#) to use it in production. You can use your existing `es2015` preset, install `babel-plugin-transform-object-rest-spread` and add it individually to the `plugins` array in your `.babelrc`.

```
{
  "presets": ["es2015"],
  "plugins": ["transform-object-rest-spread"]
}
```

Note that this is still an experimental language feature proposal so it may change in the future. Nevertheless some large projects such as [React Native](#) already use it extensively so it is safe to say that there will be a good automated migration path if it changes.

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 from 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 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) {
    let 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, 'todo')
export const editTodo = makeActionCreator(EDIT_TODO, 'id', 'todo')
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:
    let { 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)
}

componentWillReceiveProps(nextProps) {
  if (nextProps.userId !== this.props.userId) {
    this.loadData(nextProps.userId)
  }
}

render() {
  if (this.props.isFetching) {
    return <p>Loading...</p>
  }

  let posts = this.props.posts.map(post =>
    <Post post={post} key={post.id} />
  )

  return <div>{posts}</div>
}

export default connect(state => ({
  posts: state.posts
}))(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) {
    let { 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))
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.userId !== this.props.userId) {
      this.props.dispatch(loadPosts(nextProps.userId))
    }
  }

  render() {
    if (this.props.isFetching) {
      return <p>Loading...</p>
    }

    let posts = this.props.posts.map(post =>
      <Post post={post} key={post.id} />
    )

    return <div>{posts}</div>
  }
}

export default connect(state => ({
  posts: state.posts
}))(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 fetch 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](#):

```

let _todos = []

const TodoStore = Object.assign({}, EventEmitter.prototype, {
  getAll() {
    return _todos
  }
})

AppDispatcher.register(function (action) {
  switch (action.type) {
    case ActionTypes.ADD_TODO:
      let 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:
      let 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) {
    let 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) => state`, 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.

`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

// 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, initialState) { /* ... */ }

app.listen(port)
```

Handling the Request

The first thing that we need to do on every request is 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 initialState = store.getState()

  // Send the rendered page back to the client
  res.send(renderFullPage(html, initialState))
}
```

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 `initialState` to `window.__INITIAL_STATE__`.

The `initialState` will then be available on the client side by accessing `window.__INITIAL_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, initialState) {
  return `

<!doctype html>
<html>
  <head>
    <title>Redux Universal Example</title>
  </head>
  <body>
    <div id="root">${html}</div>
    <script>
      window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}
    </script>
    <script src="/static/bundle.js"></script>
  </body>
</html>
`}

}
```

Note on String Interpolation Syntax

In the example above, we use ES6 [template strings](#) syntax. It lets us write multiline strings and interpolate values, but it requires ES6 support. If you'd like to write your Node code using ES6, check out [Babel require hook](#) documentation. Or you can just keep writing ES5 code.

The Client Side

The client side is very straightforward. All we need to do is grab the initial [state](#) from `window.__INITIAL_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 { render } 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 injected into server-generated HTML
const initialState = window.__INITIAL_STATE__

// Create Redux store with initial state
const store = createStore(counterApp, initialState)

render(
  <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 `dist/bundle.js`.

When the page loads, the bundle file will be started up and `ReactDOM.render()` will hook into the `data-react-id` attributes from 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 initialState = { counter }

  // Create a new Redux store instance
  const store = createStore(counterApp, initialState)

  // 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 `__INITIAL_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 initialState = { counter }

    // Create a new Redux store instance
    const store = createStore(counterApp, initialState)

    // 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 [validator.js](#).

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 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 [Mocha](#) 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 mocha
```

To use it together with [Babel](#), you will need to install `babel-register` :

```
npm install --save-dev babel-register
```

and configure it to use ES2015 features in `.babelrc` :

```
{
  "presets": ["es2015"]
}
```

Then, add this to `scripts` in your `package.json` :

```
{
  ...
  "scripts": {
    ...
    "test": "mocha --compilers js:babel-register --recursive",
    "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 expect from 'expect'
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 [nock](#) to mock the HTTP requests.

Example

Writing Tests

```
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(json => dispatch(fetchTodosSuccess(json.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/counter'
import * as types from '../../constants/ActionTypes'
import nock from 'nock'
import expect from 'expect'; // You can use any testing library

const middlewares = [ thunk ]
const mockStore = configureMockStore(middlewares)

describe('async actions', () => {
  afterEach(() => {
    nock.cleanAll()
  })

  it('creates FETCH_TODOS_SUCCESS when fetching todos has been done', () => {
    nock('http://example.com/')
      .get('/todos')
      .reply(200, { body: { todos: ['do something'] } })

    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

Writing Tests

```
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 expect from 'expect'
import reducer from '../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 [React Test Utilities](#):

```
npm install --save-dev react-addons-test-utils
```

To test the components we make a `setup()` helper that passes the stubbed callbacks as props and renders the component with [React shallow renderer](#). This lets individual tests assert on whether the callbacks were called when expected.

Example

```
import React, { PropTypes, Component } from 'react'
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:

Writing Tests

```
import expect from 'expect'
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Header from '../../components/Header'
import TodoTextInput from '../../components/TodoTextInput'

function setup() {
  let props = {
    addTodo: expect.createSpy()
  }

  let renderer = TestUtils.createRenderer()
  renderer.render(<Header {...props} />)
  let output = renderer.getRenderOutput()

  return {
    props,
    output,
    renderer
  }
}

describe('components', () => {
  describe('Header', () => {
    it('should render correctly', () => {
      const { output } = setup()

      expect(output.type).toBe('header')
      expect(output.props.className).toBe('header')

      let [ h1, input ] = output.props.children

      expect(h1.type).toBe('h1')
      expect(h1.props.children).toBe('todos')

      expect(input.type).toBe(TodoTextInput)
      expect(input.props.newTodo).toBe(true)
      expect(input.props.placeholder).toBe('What needs to be done?')
    })
  })

  it('should call addTodo if length of text is greater than 0', () => {
    const { output, props } = setup()
    let input = output.props.children[1]
    input.props.onSave('')
    expect(props.addTodo.calls.length).toBe(0)
    input.props.onSave('Use Redux')
    expect(props.addTodo.calls.length).toBe(1)
  })
})
})
```

Fixing Broken `setState()` in older React versions

In React `<= 0.13, 0.14.4` and `0.14.5`, Shallow rendering [used to throw an error if `setState` is called](#). React seems to expect that, if you use `setState`, the DOM is available. To work around the issue, we use `jsdom` so React doesn't throw the exception when the DOM isn't available. Here's how to [set it up](#):

```
npm install --save-dev jsdom
```

Then create a `setup.js` file in your test directory:

```
import { jsdom } from 'jsdom'

global.document = jsdom('
<!doctype html>
<html><body></body></html>
')
global.window = document.defaultView
global.navigator = global.window.navigator
```

It's important that this code is evaluated *before* React is imported. To ensure this, modify your `mocha` command to include `--require ./test/setup.js` in the options in your `package.json`:

```
{
  ...
  "scripts": {
    ...
    "test": "mocha --compilers js:babel-register --recursive --require ./test/setup.js",
  },
  ...
}
```

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

```
import expect from 'expect'
import * as types from '../../constants/ActionTypes'
import singleDispatch from '../../middleware/singleDispatch'

const createFakeStore = fakeData => ({
  getState() {
    return fakeData
  }
})

const dispatchWithStoreOf = (storeData, action) => {
  let dispatched = null
  const dispatch = singleDispatch(createFakeStore(storeData))(actionAttempt => dispatched =
actionAttempt)
  dispatch(action)
  return dispatched
};

describe('middleware', () => {
  it('should dispatch if store is empty', () => {
    const action = {
      type: types.ADD_TODO
    }

    expect(
      dispatchWithStoreOf({}, action)
    ).toEqual(action)
  })

  it('should not dispatch if store already has type', () => {
    const action = {
      type: types.ADD_TODO
    }

    expect(
      dispatchWithStoreOf({
        [types.ADD_TODO]: 'dispatched'
      }, action)
    ).toNotExist()
  })
})
```

Glossary

- **React Test Utils:** Test Utilities for React.

- [jsdom](#): A plain JavaScript implementation of the DOM API. jsdom allows us to run the tests without browser.
- [Shallow rendering](#): Shallow rendering lets you instantiate a component and get the result of its `render` method just a single level deep instead of rendering components recursively to a DOM. The result of shallow rendering is a [ReactElement](#). That means it is possible to access its children, props and test if it works as expected. This also means that changing a child component won't affect the tests for parent component.

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 mutated 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

This section introduces an hypothetical extension to our app that allows it to support multiple Todo Lists. Please note that a full implementation of this extension requires changes to the reducers, components, actions etc. that aren't directly relevant to the topics discussed and have been omitted for brevity.

So far we have only seen selectors receive the Redux `store state` as an argument, but a selector can receive props too.

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 Footer from './Footer'
import AddTodo from '../containers/AddTodo'
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 let's 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(
  mapStateToProps,
  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.

Undo Redo

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:

Implementing Undo History

```
{  
  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:

Implementing Undo History

```
{  
  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:

Implementing Undo History

```
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

Implementing Undo History

```
/* ... */

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!

Redux FAQ

Table of Contents

- **General**
 - [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?](#)
- **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?](#)
 - [Should I dispatch multiple actions in a row from one action creator?](#)
- **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?](#)

- **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?

- **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?

General

When should I use Redux?

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.

Further information

Documentation

- [Introduction: Motivation](#)

Discussions

- [React How-To](#)
- [Twitter: Don't use Redux until...](#)
- [The Case for Flux](#)
- [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?](#)

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.

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!

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](#)

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.

Further information

Documentation

- [Recipes: Reducing Boilerplate](#)

Discussions

- #883: take away the huge switch block
- #1167: Reducer without switch

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](#). Find a balance that works for you, and go with it.

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.

Further information

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?
- 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?

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](#).

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

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-form](#) can provide help and abstractions in managing normalized data.

Further information

Documentation

- Advanced: Async Actions
- Examples: Real World example

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
- Twitter: state shape should be normalized

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](#) and [redux-subscribe](#) 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

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 keep 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](#)

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](#)

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
- Redux Side-Effects and You
- Pure functionality and side effects in Redux

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. If you use React, note that you can improve performance of multiple synchronous dispatches by wrapping them in

`ReactDOM.unstable_batchedUpdates()` , but this API is experimental and may be removed in any React release so don't rely on it too heavily. Take a look at [redux-batched-actions](#) that lets you dispatch several actions as if it was one and "unpack" them in the [reducer](#), and [redux-batched-subscribe](#) which lets you debounce subscriber calls for multiple dispatches.

Further information

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?

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.

Further information

Discussions

- #839: Emphasize defining selectors alongside reducers
- #943: Reducer querying
- React Boilerplate #27: Application Structure
- Stack Overflow: How to structure Redux components/containers
- 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

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.

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

Discussions

- #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?

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.

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

Discussions

- [#310: Who uses Redux?](#)
- [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](#)

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 independent sub-reducers, and even have 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 sub-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 `reduxr-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.

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

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 a number of community add-ons that provide batching of subscription notifications when multiple actions are dispatched, such as [redux-batched-subscribe](#) and [redux-batched-actions](#).

Further information

Discussions

- [#125: Strategy for avoiding cascading renders](#)
- [#542: Idea: batching actions](#)
- [#911: Batching actions](#)
- [React Redux #263: Huge performance issue when dispatching hundreds of actions](#)

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?](#)
- [Reddit: What's the best place to keep initial state?](#)

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, 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](#)

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](#)
- [Pros and Cons of Using Immutability with React](#)

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

Discussions

- [Stack Overflow: Can a React Redux app scale as well as Backbone?](#)
- [React.js pure render performance anti-pattern](#)
- [A Deep Dive into React Perf Debugging](#)

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

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 that 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 general, try to find a balance between understandable data flow and areas of responsibility with your components.

Further information

Documentation

- [Basics: Usage with React](#)

Discussions

- [Presentational and Container Components](#)
- [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?](#)

Miscellaneous

Are there any larger, “real” Redux projects?

The Redux “examples” folder has several sample projects of varying complexity, including a “real-world” example. While many companies are using Redux, most of their applications are proprietary and not available. A large number of Redux-related projects can be found on Github, such as [SoundRedux](#).

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

Discussions

- [Authentication with JWT by Auth0](#)

- [Tips to Handle Authentication in Redux](#)
- [react-redux-jwt-auth-example](#)
- [redux-auth](#)

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.

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, initialState: ?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\(reducer, initialState\)`](#) 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, \[initialState\]\)](#)
- [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, [initialState], [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. `[initialState] (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
  }
}

let 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'
let 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 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 feel that the callback needs to be invoked with the current `state`, you might want to [convert the store to an Observable](#) or [write a custom `observeStore` utility instead](#).

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)
  }
}
}

let unsubscribe = store.subscribe(handleChange)
handleChange()
```

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. `reducer` (*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 shape of the state object matches the keys of the passed reducers**.

Consequently, the [state](#) object will look like this:

```
{  
  reducer1: ...  
  reducer2: ...  
}
```

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.

Example

`reducers/todos.js`

combineReducers

```
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'

let 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(...middlewares)

[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

- `...middlewares (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.
    let 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
  }
}

let 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'

let reducer = combineReducers(reducers)
// applyMiddleware supercharges createStore with middleware:
let 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)),
    )
  }
}
```

applyMiddleware

```
    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')
    )
  )
}
```

applyMiddleware

```
        )
    }
}

// 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)
    )
  }

  componentWillReceiveProps(nextProps) {
    if (nextProps.forPerson !== this.props.forPerson) {
      this.props.dispatch(
        makeASandwichWithSecretSauce(nextProps.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') {
  let c = require('some-debug-middleware');
  let d = require('another-debug-middleware');
  middleware = [ ...middleware, c, d ];
}

const store = createStore(
  reducer,
  initialState,
  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 a single function as the first argument, and get a 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

bindActionCreators

```
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 {
  componentDidMount() {
    // Injected by react-redux:
    let { dispatch } = this.props

    // Note: this won't work:
    // TodoActionCreators.addTodo('Use Redux')

    // You're just calling a function that creates an action.
    // You must dispatch the action, too!

    // This will work:
    let action = TodoActionCreators.addTodo('Use Redux')
    dispatch(action)
  }

  render() {
    // Injected by react-redux:
    let { todos, dispatch } = this.props

    // Here's a good use case for bindActionCreators:
    // You want a child component to be completely unaware of Redux.
  }
}
```

```

let boundActionCreators = bindActionCreators(TodoActionCreators, dispatch)
console.log(boundActionCreators)
// {
//   addTodo: Function,
//   removeTodo: Function
// }

return (
  <TodoList todos={todos}
    {...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` arguments 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, combineReducers, applyMiddleware, compose } from 'redux'
import thunk from 'redux-thunk'
import DevTools from './containers/DevTools'
import reducer from '../reducers/index'

const store = createStore(
  reducer,
  compose(
    applyMiddleware(thunk),
    DevTools.instrument()
  )
)
```

Tips

compose

- 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](#).

Recent ActivityTopFixed

▲6There is no official documentation on routing1

▲57Official CRUD example11

▲1

238