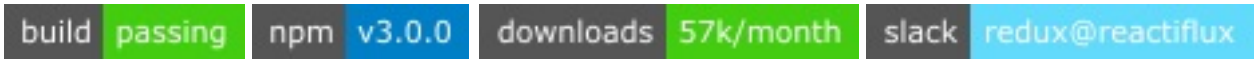
Table of Contents

Redux

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

You can use Redux together with React, or with any other view library.
It is tiny (2kB) and has no dependencies.



Testimonials

> "Love what you're doing with Redux"
> Jing Chen, creator of Flux
>
> "I asked for comments on Redux in FB's internal JS discussion group, and it was universally praised. Really awesome work."
> Bill Fisher, creator of Flux
>
> "It's cool that you are inventing a better Flux by not doing Flux at all."
> André Staltz, creator of Cycle

Developer Experience

I wrote Redux while working on my React Europe talk called "Hot Reloading with Time Travel". My goal was to create a state management library with minimal API but completely predictable behavior, so it is possible to implement logging, hot reloading, time travel, universal apps, record and replay, without any buy-in from the developer.

Influences

Redux evolves the ideas of Flux, but avoids its complexity by taking cues from Elm.

Whether you have used them or not, Redux takes a few minutes to get started with.

Installation

To install the stable version:

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

Most likely, you'll also need the React bindings and the developer tools.

```
npm install --save react-redux
npm install --save-dev redux-devtools
```

This assumes that you're using npm package manager with a module bundler like Webpack or Browserify to consume CommonJS modules.

If you don't yet use npm or a modern module bundler, and would rather prefer a single-file UMD build that makes `Redux` available as a global object, you can grab a pre-built version from cdnjs. We *don't* recommend this approach for any serious application, as most of the libraries complementary to Redux are only available on npm.

The Gist

The whole state of your app is stored in an object tree inside a single *store*.
The only way to change the state tree is to emit an *action*, an object describing what happened.
To specify how the actions transform the state tree, you write pure *reducers*.

That's it!

```
import { createStore } from 'redux';

/**
 * This is a reducer, a pure function with (state, action) => state signature.
 * It describes how an action transforms the state into the next state.
 *
 * The shape of the state is up to you: it can be a primitive, an array, an object,
 * or even an Immutable.js data structure. The only important part is that you should
 * not mutate the state object, but return a new object if the state changes.
```

```
 *
 * In this example, we use a `switch` statement and strings, but you can use a helper
 * follows a different convention (such as function maps) if it makes sense for your
 */
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  default:
    return state;
  }
}

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

// You can subscribe to the updates manually, or use bindings to your view layer.
store.subscribe(() =>
  console.log(store.getState())
);

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

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

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. This is exactly like there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like an overkill for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused

it. You can record user sessions and reproduce them just by replaying every action.

Documentation

- Introduction
- Basics
- Advanced
- Recipes
- Troubleshooting
- Glossary
- API Reference

For PDF, ePub, and MOBI exports for offline reading, and instructions on how to create them, please see: paulwittmann/redux-offline-docs.

Examples

- Counter (source)
- TodoMVC (source)
- Async (source)
- Real World (source)

If you're new to the NPM ecosystem and have troubles getting a project up and running, or aren't sure where to paste the gist above, check out simplest-redux-example that uses Redux together with React and Browserify.

Discussion

Join the **#redux** channel of the Reactiflux Slack community.

Thanks

- The Elm Architecture for a great intro to modeling state updates with reducers;
- Turning the database inside-out for blowing my mind;
- Developing ClojureScript with Figwheel for convincing me that re-evaluation should "just work";
- Webpack for Hot Module Replacement;
- Flummox for teaching me to approach Flux without boilerplate or singletons;

- disto for a proof of concept of hot reloadable Stores;
- NuclearJS for proving this architecture can be performant;
- Om for popularizing the idea of a single state atom;
- Cycle for showing how often a function is the best tool;
- React for the pragmatic innovation.

Special thanks to Jamie Paton for handing over the `redux` NPM package name.

Change Log

This project adheres to Semantic Versioning.
Every release, along with the migration instructions, is documented on the Github Releases page.

Patrons

The work on Redux was funded by the community.
Meet some of the outstanding companies that made it possible:

- Webflow
- Chess iX

See the full list of Redux patrons.

License

MIT

Introduction

Motivation

As the requirements to JavaScript single-page applications get more sophisticated, **more state needs to be managed** by the JavaScript code than ever before. This state may include server responses, cached data, and data created locally, but not yet persisted to the server. It also includes the UI state, such as the active route, the selected tab, whether to show a spinner or pagination controls, and so on.

Managing ever-changing state is hard. If a model can update another model, then a view can update a model that updates another model, and this, in turn, might cause another view to update. At some point you no longer know what happens in your app. **You no longer control when, why, and how the state is updated.** When the 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**, such as handling optimistic updates, rendering on the server, fetching data before performing route transitions, and so on. We the front-end developers are finding ourselves surrounded by the complexity we never had to deal with before. Is it time we give up?

A lot of this complexity comes from the fact that **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 are a mess. Libraries like React attempt to solve this problem in the view layer by removing asynchrony and direct DOM manipulation. However, React leaves managing the state of your data up to you.

Following 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 inside a single store.

This makes it easy to create universal apps. The state from the server can be serialized and hydrated into the client with no extra coding effort. It is easier to debug an application when there is a single state tree. You can also persist your app's state in development for a faster development cycle. And with a single state tree, you get previously difficult functionality like Undo/Redo for free.

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

{
  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 the views or the network callbacks never write directly to the state, and instead express the 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. Actions are just plain objects, so 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'
});
```

Mutations are written as 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, but as your app grows, you can split it 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.slice(0, action.index),
      Object.assign({}, state[action.index], {
        completed: true
      }),
      ...state.slice(action.index + 1)
    ];
  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 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 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 severely discouraged. You should always return a new object, which is easy with the object spread syntax proposed for ES7 and implemented in Babel, 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 created by Evan Czaplicki. It enforces a "model view update" architecture, where the update has the following signature: `(state, action) => state`. Technically, Elm "updaters" are equivalent to the reducers in Redux.

But 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 to 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.

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.

Bindings

- react-redux — React
- ng-redux — Angular

Middleware

- redux-thunk — The easiest way to write async action creators
- redux-promise — FSA-compliant promise middleware
- 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

Components

- redux-form — Keep React form state in Redux

Store enhancers

- redux-batched-subscribe — Customize batching and debouncing calls to the store subscribers

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-transducers — Transducer utilities for Redux
- redux-immutablejs - Integration tools between Redux and Immutable

Developer Tools

- redux-devtools — An action logger with time travel UI, hot reloading and error handling for the reducers, first demoed at React Europe

Tutorials and Articles

- redux-tutorial — Learn how to use Redux step by step
- What the Flux?! Let's Redux. — An intro to 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

More

Awesome Redux is an extensive list of Redux-related repositories.

Examples

Redux is distributed with a few examples in its source code.

> **Note on Copying**
>
> If you copy Redux examples outside their folders, you can delete some lines at the end of their `webpack.config.js` files. They follow a “You can safely delete these lines in your project.” comment.

Counter

Run the Counter example:

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

cd redux/examples/counter
npm install
npm start

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

It covers:

- Basic Redux flow
- Testing

TodoMVC

Run the TodoMVC example:

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

cd redux/examples/todomvc
npm install
npm start
```

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

It covers:

- Redux flow with two reducers
- Updating nested data
- Testing

Async

Run the Async example:

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

cd redux/examples/async
npm install
npm start

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

It covers:

- Basic async Redux flow with redux-thunk
- Caching responses and showing a spinner while data is fetching
- Invalidating the cached data

Universal

Run the Universal example:

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

cd redux/examples/universal
npm install
npm start & npm run client

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

It covers:

- Universal rendering with Redux and React
- Prefetching state based on input and via asynchronous fetches.
- Passing state from the server to the client

Real World

Run the Real World example:

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

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

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

It covers:

- Real-world async Redux flow
- Keeping entities in a normalized entity cache
- A custom middleware for API calls
- Caching responses and showing a spinner while data is fetching
- Pagination
- Routing

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

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:

```
{
  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: COMPLETE_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 traditional Flux implementations, action creators often trigger a dispatch when invoked, like so:

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

By contrast, in Redux action creators are **pure functions** with zero side-effects. They simply return an action:

```
function addTodo(text) {
  return {
```

```
    type: ADD_TODO,
    text
  };
}
```

This makes them more portable and easier to test. To actually initiate a dispatch, pass the result to the `dispatch()` function:

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

Or create a **bound action creator** that automatically dispatches:

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

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.

Source Code

`actions.js`

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO';
export const COMPLETE_TODO = 'COMPLETE_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 completeTodo(index) {
  return { type: COMPLETE_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!

> **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 actions in the advanced tutorial to learn how to handle AJAX responses and compose action creators into async control flow.

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;
- Calling 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 experimental object spread syntax proposed for ES7 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 `COMPLETE_TODO` handler shouldn't come as a complete surprise:

```
case COMPLETE_TODO:
  return Object.assign({}, state, {
    todos: [
      ...state.todos.slice(0, action.index),
      Object.assign({}, state.todos[action.index], {
        completed: true
      }),
      ...state.todos.slice(action.index + 1)
    ]
  });
```

Because we want to update a specific item in the array without resorting to mutations, we have to slice it before and after the item. 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 COMPLETE_TODO:
    return Object.assign({}, state, {
      todos: [
        ...state.todos.slice(0, action.index),
        Object.assign({}, state.todos[action.index], {
          completed: true
        }),
        ...state.todos.slice(action.index + 1)
```

```
      ]
    });
  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 COMPLETE_TODO:
    return [
      ...state.slice(0, action.index),
      Object.assign({}, state[action.index], {
        completed: true
      }),
      ...state.slice(action.index + 1)
    ];
  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 COMPLETE_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 COMPLETE_TODO:
    return [
      ...state.slice(0, action.index),
      Object.assign({}, state[action.index], {
        completed: true
      }),
      ...state.slice(action.index + 1)
    ];
  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, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './
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 COMPLETE_TODO:
    return [
      ...state.slice(0, action.index),
      Object.assign({}, state[action.index], {
        completed: true
      }),
      ...state.slice(action.index + 1)
    ];
  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)`.

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, completeTodo, setVisibilityFilter, VisibilityFilters } from './acti

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

// Every time the state changes, log it
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(completeTodo(0));
store.dispatch(completeTodo(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: 'Megan' } };
   { 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
```

Smart and Dumb Components

React bindings for Redux embrace the idea of separating “smart” and “dumb” components.

It is advisable that only top-level components of your app (such as route handlers) are aware of Redux. Components below them should be “dumb” and receive all data via props.

	“Smart” Components	**“Dumb” Components**
Location	Top level, route handlers	Middle and leaf components
Aware of Redux	Yes	No
To read data	Subscribe to Redux state	Read data from props
To change data	Dispatch Redux actions	Invoke callbacks from props

In this todo app, we will only have a single “smart” component at the top of our view hierarchy. In more complex apps, you might have several of them. While you may nest “smart” components, we suggest that you pass props down whenever possible.

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 / only incomplete todos.

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

- `AddTodo` is an input field with a button.
  - `onAddClick(text: string)` is a callback to invoke when a button is pressed.
- `TodoList` is a list showing visible todos.
  - `todos: Array` is an array of todo items with `{ text, completed }` shape.
  - `onTodoClick(index: 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.
- `Footer` is a component where we let user change visible todo filter.
  - `filter: string` is the current filter: `'SHOW_ALL'`, `'SHOW_COMPLETED'` or `'SHOW_ACTIVE'`.
  - `onFilterChange(nextFilter: string)` : Callback to invoke when user chooses a different filter.

These are all “dumb” components. They 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.

Let's write them! We don't need to think about binding to Redux yet. You can just give them fake data while you experiment until they render correctly.

Dumb Components

These are all normal React components, so we won't stop to examine them in detail. Here they go:

`components/AddTodo.js`

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

export default class AddTodo extends Component {
  render() {
    return (
      <div>
        <input type='text' ref='input' />
        <button onClick={e => this.handleClick(e)}>
          Add
        </button>
      </div>
    );
  }

  handleClick(e) {
    const node = findDOMNode(this.refs.input);
    const text = node.value.trim();
    this.props.onAddClick(text);
    node.value = '';
  }
}

AddTodo.propTypes = {
  onAddClick: PropTypes.func.isRequired
};
```

`components/Todo.js`

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

export default class Todo extends Component {
  render() {
    return (
      <li
        onClick={this.props.onClick}
        style={{
```

```
          textDecoration: this.props.completed ? 'line-through' : 'none',
          cursor: this.props.completed ? 'default' : 'pointer'
        }}>
        {this.props.text}
      </li>
    );
  }
}

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

components/TodoList.js

```
import React, { Component, PropTypes } from 'react';
import Todo from './Todo';

export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.todos.map((todo, index) =>
          <Todo {...todo}
                key={index}
                onClick={() => this.props.onTodoClick(index)} />
        )}
      </ul>
    );
  }
}

TodoList.propTypes = {
  onTodoClick: PropTypes.func.isRequired,
  todos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired
};
```

components/Footer.js

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

export default class Footer extends Component {
  renderFilter(filter, name) {
    if (filter === this.props.filter) {
```

```
        return name;
      }

      return (
        <a href='#' onClick={e => {
          e.preventDefault();
          this.props.onFilterChange(filter);
        }}>
          {name}
        </a>
      );
    }

    render() {
      return (
        <p>
          Show:
          {' '}
          {this.renderFilter('SHOW_ALL', 'All')}
          {', '}
          {this.renderFilter('SHOW_COMPLETED', 'Completed')}
          {', '}
          {this.renderFilter('SHOW_ACTIVE', 'Active')}
          .
        </p>
      );
    }
  }

  Footer.propTypes = {
    onFilterChange: PropTypes.func.isRequired,
    filter: PropTypes.oneOf([
      'SHOW_ALL',
      'SHOW_COMPLETED',
      'SHOW_ACTIVE'
    ]).isRequired
  };
```

That's it! We can verify that they work correctly by writing a dummy `App` to render them:

`containers/App.js`

```
  import React, { Component } from 'react';
  import AddTodo from '../components/AddTodo';
  import TodoList from '../components/TodoList';
  import Footer from '../components/Footer';

  export default class App extends Component {
    render() {
      return (
        <div>
          <AddTodo
```

```
            onAddClick={text =>
              console.log('add todo', text)
            } />
          <TodoList
            todos={[{
              text: 'Use Redux',
              completed: true
            }, {
              text: 'Learn to connect it to React',
              completed: false
            }]}
            onTodoClick={todo =>
              console.log('todo clicked', todo)
            } />
          <Footer
            filter='SHOW_ALL'
            onFilterChange={filter =>
              console.log('filter change', filter)
            } />
        </div>
      );
    }
  }
```

This is what I see when I render `<App />`:

By itself, it's not very interesting. Let's connect it to Redux!

Connecting to Redux

We need to make two changes to connect our `App` component to Redux and make it dispatch actions and read state from the Redux store.

First, we need to import `Provider` from `react-redux`, which we installed earlier, and **wrap the root component in `<Provider>`** before rendering.

`index.js`

```
import React from 'react';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import App from './containers/App';
import todoApp from './reducers';

let store = createStore(todoApp);

let rootElement = document.getElementById('root');
React.render(
  // The child must be wrapped in a function
  // to work around an issue in React 0.13.
  <Provider store={store}>
    {() => <App />}
  </Provider>,
  rootElement
);
```

This makes our store instance available to the components below. (Internally, this is done via React's undocumented “context” feature, but it’s not exposed directly in the API so don’t worry about it.)

Then, we **wrap the components we want to connect to Redux with the `connect()` function from `react-redux`**. Try to only do this for a top-level component, or route handlers. While technically you can `connect()` any component in your app to Redux store, avoid doing this too deeply, because it will make the data flow harder to trace.

Any component wrapped with `connect()` call will receive a `dispatch` function as a prop, and any state it needs from the global state. The only argument to `connect()` is a function we call a **selector**. This function takes the global Redux store’s state, and returns the props you need for the component. In the simplest case, you can just return the `state` given to you, but you may also wish to transform it first.

To make performant memoized transformations with composable selectors, check out reselect. In this example, we won’t use it, but it works great for larger apps.

`containers/App.js`

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilters } from '../act
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';
```

```
class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    );
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  })),
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
};

function selectTodos(todos, filter) {
  switch (filter) {
  case VisibilityFilters.SHOW_ALL:
    return todos;
  case VisibilityFilters.SHOW_COMPLETED:
    return todos.filter(todo => todo.completed);
  case VisibilityFilters.SHOW_ACTIVE:
    return todos.filter(todo => !todo.completed);
  }
}

// Which props do we want to inject, given the global state?
// Note: use https://github.com/faassen/reselect for better performance.
function select(state) {
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}
```

```
// Wrap the component to inject dispatch and state into it
export default connect(select)(App);
```

That's it! The tiny todo app now functions correctly.

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 { createStore } from 'redux';
import { Provider } from 'react-redux';
import App from './containers/App';
import todoApp from './reducers';

let store = createStore(todoApp);

let rootElement = document.getElementById('root');
React.render(
  // The child must be wrapped in a function
  // to work around an issue in React 0.13.
  <Provider store={store}>
    {() => <App />}
  </Provider>,
  rootElement
);
```

Action Creators and Constants

`actions.js`

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO';
export const COMPLETE_TODO = 'COMPLETE_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 completeTodo(index) {
  return { type: COMPLETE_TODO, index };
}

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

Reducers

`reducers.js`

```
import { combineReducers } from 'redux';
import { ADD_TODO, COMPLETE_TODO, SET_VISIBILITY_FILTER, VisibilityFilters } from './
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 COMPLETE_TODO:
    return [
      ...state.slice(0, action.index),
```

```
        Object.assign({}, state[action.index], {
          completed: true
        }),
        ...state.slice(action.index + 1)
      ];
    default:
      return state;
    }
  }

  const todoApp = combineReducers({
    visibilityFilter,
    todos
  });

  export default todoApp;
```

Smart Components

`containers/App.js`

```
  import React, { Component, PropTypes } from 'react';
  import { connect } from 'react-redux';
  import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilters } from '../act
  import AddTodo from '../components/AddTodo';
  import TodoList from '../components/TodoList';
  import Footer from '../components/Footer';

  class App extends Component {
    render() {
      // Injected by connect() call:
      const { dispatch, visibleTodos, visibilityFilter } = this.props;
      return (
        <div>
          <AddTodo
            onAddClick={text =>
              dispatch(addTodo(text))
            } />
          <TodoList
            todos={visibleTodos}
            onTodoClick={index =>
              dispatch(completeTodo(index))
            } />
          <Footer
            filter={visibilityFilter}
            onFilterChange={nextFilter =>
              dispatch(setVisibilityFilter(nextFilter))
            } />
        </div>
```

```
    );
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  })),
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
};

function selectTodos(todos, filter) {
  switch (filter) {
  case VisibilityFilters.SHOW_ALL:
    return todos;
  case VisibilityFilters.SHOW_COMPLETED:
    return todos.filter(todo => todo.completed);
  case VisibilityFilters.SHOW_ACTIVE:
    return todos.filter(todo => !todo.completed);
  }
}

// Which props do we want to inject, given the global state?
// Note: use https://github.com/faassen/reselect for better performance.
function select(state) {
  return {
    visibleTodos: selectTodos(state.todos, state.visibilityFilter),
    visibilityFilter: state.visibilityFilter
  };
}

// Wrap the component to inject dispatch and state into it
export default connect(select)(App);
```

Dumb Components

`components/AddTodo.js`

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

export default class AddTodo extends Component {
  render() {
    return (
      <div>
```

```
          <input type='text' ref='input' />
          <button onClick={(e) => this.handleClick(e)}>
            Add
          </button>
        </div>
      );
    }

    handleClick(e) {
      const node = findDOMNode(this.refs.input);
      const text = node.value.trim();
      this.props.onAddClick(text);
      node.value = '';
    }
  }

  AddTodo.propTypes = {
    onAddClick: PropTypes.func.isRequired
  };
```

components/Footer.js

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

  export default class Footer extends Component {
    renderFilter(filter, name) {
      if (filter === this.props.filter) {
        return name;
      }

      return (
        <a href='#' onClick={e => {
          e.preventDefault();
          this.props.onFilterChange(filter);
        }}>
          {name}
        </a>
      );
    }

    render() {
      return (
        <p>
          Show:
          {' '}
          {this.renderFilter('SHOW_ALL', 'All')}
          {', '}
          {this.renderFilter('SHOW_COMPLETED', 'Completed')}
          {', '}
          {this.renderFilter('SHOW_ACTIVE', 'Active')}
          .
        </p>
```

```
    );
  }
}

Footer.propTypes = {
  onFilterChange: PropTypes.func.isRequired,
  filter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
};
```

components/Todo.js

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

export default class Todo extends Component {
  render() {
    return (
      <li
        onClick={this.props.onClick}
        style={{
          textDecoration: this.props.completed ? 'line-through' : 'none',
          cursor: this.props.completed ? 'default' : 'pointer'
        }}>
        {this.props.text}
      </li>
    );
  }
}

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

components/TodoList.js

```
import React, { Component, PropTypes } from 'react';
import Todo from './Todo';

export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.todos.map((todo, index) =>
          <Todo {...todo}
```

```
                key={index}
                onClick={() => this.props.onTodoClick(index)} />
        )}
      </ul>
    );
  }
}

TodoList.propTypes = {
  onTodoClick: PropTypes.func.isRequired,
  todos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  }).isRequired).isRequired
};
```

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

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`. Maybe, some reducers will also 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 reddit to display:

```
export const SELECT_REDDIT = 'SELECT_REDDIT';

export function selectReddit(reddit) {
  return {
    type: SELECT_REDDIT,
    reddit
  };
}
```

They can also press a "refresh" button to update it:

```
export const INVALIDATE_REDDIT = 'INVALIDATE_REDDIT';

export function invalidateReddit(reddit) {
  return {
    type: INVALIDATE_REDDIT,
```

```
    reddit
  };
}
```

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 reddit, we will dispatch a `REQUEST_POSTS` action:

```
export const REQUEST_POSTS = 'REQUEST_POSTS';

export function requestPosts(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit
  };
}
```

It is important for it to be separate from `SELECT_REDDIT` or `INVALIDATE_REDDIT`. 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 reddits, 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(reddit, json) {
  return {
    type: RECEIVE_POSTS,
    reddit,
    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:

```
{
  selectedReddit: 'frontend',
  postsByReddit: {
    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:
>
> ```
> {
>   selectedReddit: '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 Architec
          author: 2
        }
      }
    },
    postsByReddit: {
      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_REDDIT, INVALIDATE_REDDIT,
  REQUEST_POSTS, RECEIVE_POSTS
} from '../actions';

function selectedReddit(state = 'reactjs', action) {
  switch (action.type) {
  case SELECT_REDDIT:
    return action.reddit;
  default:
    return state;
  }
}

function posts(state = {
  isFetching: false,
  didInvalidate: false,
  items: []
}, action) {
  switch (action.type) {
  case INVALIDATE_REDDIT:
    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 postsByReddit(state = {}, action) {
  switch (action.type) {
  case INVALIDATE_REDDIT:
  case RECEIVE_POSTS:
  case REQUEST_POSTS:
    return Object.assign({}, state, {
      [action.reddit]: posts(state[action.reddit], action)
    });
  default:
    return state;
  }
}
```

```
const rootReducer = combineReducers({
  postsByReddit,
  selectedReddit
});

export default rootReducer;
```

In this code, there are two interesting parts:

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

  ```
  return Object.assign({}, state, {
    [action.reddit]: posts(state[action.reddit], action)
  });
  ```

  is equivalent to this:

  ```
  let nextState = {};
  nextState[action.reddit] = posts(state[action.reddit], 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 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(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit
  };
}

export const RECEIVE_POSTS = 'RECEIVE_POSTS'
function receivePosts(reddit, json) {
  return {
    type: RECEIVE_POSTS,
    reddit,
    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 crea
// store.dispatch(fetchPosts('reactjs'));

export function fetchPosts(reddit) {

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

  return function (dispatch) {

    // First dispatch: the app state is updated to inform
    // that the API call is starting.

    dispatch(requestPosts(reddit));

    // 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/${reddit}.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(reddit, 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-core/polyfill';
> ```

How do we include the Redux Thunk middleware in the dispatch mechanism? We use

the `applyMiddleware()` method from Redux, as shown below:

index.js

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

const loggerMiddleware = createLogger();

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

const store = createStoreWithMiddleware(rootReducer);

store.dispatch(selectReddit('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(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit
  };
}

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

function fetchPosts(reddit) {
```

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

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

export function fetchPostsIfNeeded(reddit) {

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

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

  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), reddit)) {
      // Dispatch a thunk from thunk!
      return dispatch(fetchPosts(reddit));
    } 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

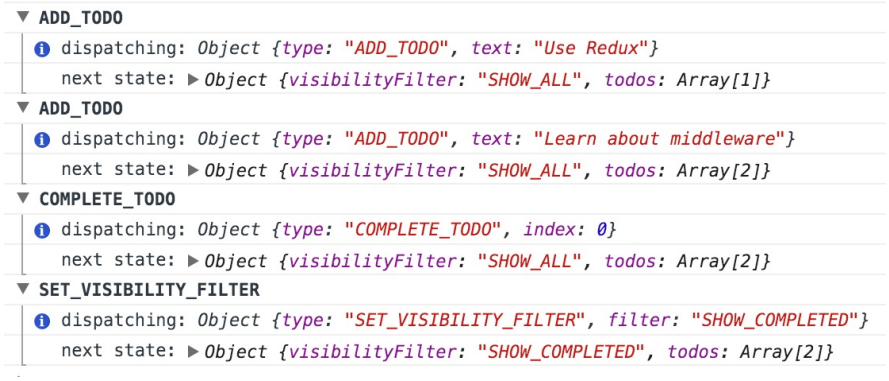
While middleware can be used for a variety of things, including asynchronous API calls, it's really important that you understand where it comes from. We'll guide you through the thought process leading to middleware, by using logging and crash reporting as examples.

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.



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`.

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';

// applyMiddleware takes createStore() and returns
// a function with a compatible API.
let createStoreWithMiddleware = applyMiddleware(logger, crashReporter)(createStore);

// Use it like you would use createStore()
let todoApp = combineReducers(reducers);
let store = createStoreWithMiddleware(todoApp);
```

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
   * Makes `dispatch` return a function to remove the action from the queue in this cas
   */
  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 createStoreWithMiddleware = applyMiddleware(
  rafScheduler,
  timeoutScheduler,
  thunk,
  vanillaPromise,
  readyStatePromise,
  logger,
  crashReporter
)(createStore);
let todoApp = combineReducers(reducers);
let store = createStoreWithMiddleware(todoApp);
```

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-core/polyfill';

import React from 'react';
import Root from './containers/Root';

React.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_REDDIT = 'SELECT_REDDIT';
export const INVALIDATE_REDDIT = 'INVALIDATE_REDDIT';

export function selectReddit(reddit) {
  return {
    type: SELECT_REDDIT,
    reddit
  };
}

export function invalidateReddit(reddit) {
  return {
    type: INVALIDATE_REDDIT,
    reddit
```

```
  };
}

function requestPosts(reddit) {
  return {
    type: REQUEST_POSTS,
    reddit
  };
}

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

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

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

export function fetchPostsIfNeeded(reddit) {
  return (dispatch, getState) => {
    if (shouldFetchPosts(getState(), reddit)) {
      return dispatch(fetchPosts(reddit));
    }
  };
}
```

Reducers

`reducers.js`

```
import { combineReducers } from 'redux';
import {
  SELECT_REDDIT, INVALIDATE_REDDIT,
  REQUEST_POSTS, RECEIVE_POSTS
} from './actions';

function selectedReddit(state = 'reactjs', action) {
  switch (action.type) {
  case SELECT_REDDIT:
    return action.reddit;
  default:
    return state;
  }
}

function posts(state = {
  isFetching: false,
  didInvalidate: false,
  items: []
}, action) {
  switch (action.type) {
  case INVALIDATE_REDDIT:
    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 postsByReddit(state = { }, action) {
  switch (action.type) {
  case INVALIDATE_REDDIT:
  case RECEIVE_POSTS:
  case REQUEST_POSTS:
    return Object.assign({}, state, {
      [action.reddit]: posts(state[action.reddit], action)
    });
  default:
    return state;
  }
}
```

```
const rootReducer = combineReducers({
  postsByReddit,
  selectedReddit
});

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();

const createStoreWithMiddleware = applyMiddleware(
  thunkMiddleware,
  loggerMiddleware
)(createStore);

export default function configureStore(initialState) {
  return createStoreWithMiddleware(rootReducer, initialState);
}
```

Smart Components

`containers/Root.js`

```
import React, { Component } from 'react';
import { Provider } from 'react-redux';
import configureStore from '../configureStore';
import AsyncApp from './AsyncApp';

const store = configureStore();

export default class Root extends Component {
  render() {
    return (
      <Provider store={store}>
        {() => <AsyncApp />}
      </Provider>
```

```
    );
  }
}
```

containers/AsyncApp.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { selectReddit, fetchPostsIfNeeded, invalidateReddit } 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, selectedReddit } = this.props;
    dispatch(fetchPostsIfNeeded(selectedReddit));
  }

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

  handleChange(nextReddit) {
    this.props.dispatch(selectReddit(nextReddit));
  }

  handleRefreshClick(e) {
    e.preventDefault();

    const { dispatch, selectedReddit } = this.props;
    dispatch(invalidateReddit(selectedReddit));
    dispatch(fetchPostsIfNeeded(selectedReddit));
  }

  render () {
    const { selectedReddit, posts, isFetching, lastUpdated } = this.props;
    return (
      <div>
        <Picker value={selectedReddit}
                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 = {
    selectedReddit: PropTypes.string.isRequired,
    posts: PropTypes.array.isRequired,
    isFetching: PropTypes.bool.isRequired,
    lastUpdated: PropTypes.number,
    dispatch: PropTypes.func.isRequired
  };

  function mapStateToProps(state) {
    const { selectedReddit, postsByReddit } = state;
    const {
      isFetching,
      lastUpdated,
      items: posts
    } = postsByReddit[selectedReddit] || {
      isFetching: true,
      items: []
    };

    return {
      selectedReddit,
      posts,
      isFetching,
      lastUpdated
    };
  }

  export default connect(mapStateToProps)(AsyncApp);
```

Dumb Components

`components/Picker.js`

```
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`

```
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
- Reducing Boilerplate
- Server Rendering
- Writing Tests
- Computing Derived Data

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` 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

Sorry, you'll need to rewrite your model layer.
It's way too different!

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 `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-action-utils and redux-actions. These can help with reducing your boilerplate code and adhering 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 './actionCreator

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 "smart" `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.users[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 function (next) {
    return function (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: response,
          type: successType
        })),
        error => dispatch(Object.assign({}, payload, {
          error: 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.users[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.posts[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 = [];

export default const TodoStore = 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();
  }
});
```

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 React.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.

```
function handleRender(req, res) {
  // Create a new Redux store instance
  const store = createStore(counterApp);

  // Render the component to a string
  const html = React.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="app">${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 { 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);

React.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 `React.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

function handleRender(req, res) {
  // Read the counter from the request, if provided
  const params = qs.parse(req.query);
  const counter = parseInt(params.counter) || 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 = React.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';

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) || 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 = React.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 `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 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.

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/App.js`

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { addTodo, completeTodo, setVisibilityFilter, VisibilityFilters } from '../act
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';

class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={this.props.visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    );
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
```

```
    })),
    visibilityFilter: PropTypes.oneOf([
      'SHOW_ALL',
      'SHOW_COMPLETED',
      'SHOW_ACTIVE'
    ]).isRequired
  };

  function selectTodos(todos, filter) {
    switch (filter) {
    case VisibilityFilters.SHOW_ALL:
      return todos;
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(todo => todo.completed);
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(todo => !todo.completed);
    }
  }

  function select(state) {
    return {
      visibleTodos: selectTodos(state.todos, state.visibilityFilter),
      visibilityFilter: state.visibilityFilter
    };
  }

  // Wrap the component to inject dispatch and state into it
  export default connect(select)(App);
```

In the above example, `select` calls `selectTodos` to calculate `visibleTodos`. This works great, but there is a drawback: `visibleTodos` 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 `select` with a memoized selector that recalculates `visibleTodos` 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 `visibleTodosSelector` to replace `select`:

`selectors/TodoSelectors.js`

```
import { createSelector } from 'reselect';
import { VisibilityFilters } from './actions';

function selectTodos(todos, filter) {
  switch (filter) {
  case VisibilityFilters.SHOW_ALL:
    return todos;
  case VisibilityFilters.SHOW_COMPLETED:
    return todos.filter(todo => todo.completed);
  case VisibilityFilters.SHOW_ACTIVE:
    return todos.filter(todo => !todo.completed);
  }
}

const visibilityFilterSelector = (state) => state.visibilityFilter;
const todosSelector = (state) => state.todos;

export const visibleTodosSelector = createSelector(
  [visibilityFilterSelector, todosSelector],
  (visibilityFilter, todos) => {
    return {
      visibleTodos: selectTodos(todos, visibilityFilter),
      visibilityFilter
    };
  }
);
```

In the example above, `visibilityFilterSelector` and `todosSelector` are input-selectors. They are created as ordinary non-memoized selector functions because they do not transform the data they select. `visibleTodosSelector` on the other hand is a memoized selector. It takes `visibilityFilterSelector` and `todosSelector` 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 `visibleTodosSelector` being used as an input-selector to a selector that further filters

the todos by keyword:

```
const keywordSelector = (state) => state.keyword;

const keywordFilterSelector = createSelector(
  [visibleTodosSelector, keywordSelector],
  (visibleTodos, keyword) => visibleTodos.filter(
    todo => todo.indexOf(keyword) > -1
  )
);
```

Connecting a Selector to the Redux Store

If you are using react-redux, you connect a memoized selector to the Redux store using `connect`:

`containers/App.js`

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
import { addTodo, completeTodo, setVisibilityFilter } from '../actions';
import AddTodo from '../components/AddTodo';
import TodoList from '../components/TodoList';
import Footer from '../components/Footer';
import { visibleTodosSelector } from '../selectors/todoSelectors.js';

class App extends Component {
  render() {
    // Injected by connect() call:
    const { dispatch, visibleTodos, visibilityFilter } = this.props;
    return (
      <div>
        <AddTodo
          onAddClick={text =>
            dispatch(addTodo(text))
          } />
        <TodoList
          todos={this.props.visibleTodos}
          onTodoClick={index =>
            dispatch(completeTodo(index))
          } />
        <Footer
          filter={visibilityFilter}
          onFilterChange={nextFilter =>
            dispatch(setVisibilityFilter(nextFilter))
          } />
      </div>
    );
```

```
  }
}

App.propTypes = {
  visibleTodos: PropTypes.arrayOf(PropTypes.shape({
    text: PropTypes.string.isRequired,
    completed: PropTypes.bool.isRequired
  })),
  visibilityFilter: PropTypes.oneOf([
    'SHOW_ALL',
    'SHOW_COMPLETED',
    'SHOW_ACTIVE'
  ]).isRequired
};

// Pass the selector to the connect component
export default connect(visibleTodosSelector)(App);
```

Writing Tests

Because most of the Redux code you write are functions, and many of them are pure, they are easy 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 DOM.

```
npm install --save-dev mocha
```

To use it together with Babel, 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);
  });
});
```

Reducers

A reducer should return the new state after applying the action to the previous state, and that's the behavior tested below.

Example

```
import { ADD_TODO } from '../constants/ActionTypes';

const initialState = [{
  text: 'Use Redux',
  completed: false,
  id: 0
}];

export default function todos(state = initialState, action) {
  switch (action.type) {
  case ADD_TODO:
    return [{
      id: state.reduce((maxId, todo) => Math.max(todo.id, maxId), -1) + 1,
      completed: false,
      text: action.text
    }, ...state];
```

```
    default:
      return state;
  }
}
```

can be tested like:

```
import 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.

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:

```
import expect from 'expect';
```

```
import jsdomReact from '../jsdomReact';
import React from 'react/addons';
import Header from '../../components/Header';
import TodoTextInput from '../../components/TodoTextInput';

const { TestUtils } = React.addons;

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', () => {
  jsdomReact();

  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()`

Shallow rendering currently throws an error if `setState` is called. React seems to expect that, if you use `setState`, DOM is available. To work around the issue, we use jsdom so React doesn't throw the exception when DOM isn't available. Here's how to set it up:

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

Then add a `jsdomReact()` helper function that looks like this:

```
import ExecutionEnvironment from 'react/lib/ExecutionEnvironment';
import jsdom from 'mocha-jsdom';

export default function jsdomReact() {
  jsdom();
  ExecutionEnvironment.canUseDOM = true;
}
```

Call it before running any component tests. Note this is a dirty workaround, and it can be removed once facebook/react#4019 is fixed.

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(mapDispatchToProps)(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 brances: 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 => dispat
  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 that ship with 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 you changing a child component won't affect the tests for parent component.

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.actions.
    state.actions.push({
      text: action.text,
      completed: false
    });
  case 'COMPLETE_TODO':
    // Wrong! This mutates state.actions[action.index].
    state.actions[action.index].completed = true;
  }

  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.slice(0, action.index),
      // Copy the object before mutating
      Object.assign({}, state[action.index], {
        completed: true
      }),
      ...state.slice(action.index + 1)
    ];
  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.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true
  }),
  ...state.slice(action.index + 1)
]

// 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 ES7 object spread proposal with Babel stage 1:

```
// Before:
return [
  ...state.slice(0, action.index),
  Object.assign({}, state[action.index], {
    completed: true
  }),
  ...state.slice(action.index + 1)
]

// After:
return [
  ...state.slice(0, action.index),
  { ...state[action.index], completed: true },
  ...state.slice(action.index + 1)
]
```

Note that experimental language features are subject to change, and it's unwise to rely on them in large codebases.

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.

Something else doesn't work

Ask around on the **#redux** Reactiflux Slack 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.

By convention, actions should 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])`

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.

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 ES7 object spread proposal with Babel stage 1.

- 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.

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.

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.

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', currer
  }
}

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 gather their results into a single state object. The shape of the state object matches the keys of the passed `reducers`.

> **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 may 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`

```
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, combineReducers } 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 is a

function that needs to be applied to `createStore`. It will return a different `createStore` which has the middleware enabled.

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 createStoreWithMiddleware = applyMiddleware(logger)(createStore);
let store = createStoreWithMiddleware(todos, ['Use Redux']);

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';

// applyMiddleware supercharges createStore with middleware:
let createStoreWithMiddleware = applyMiddleware(thunk)(createStore);

// We can use it exactly like “vanilla” createStore.
let reducer = combineReducers(reducers);
let store = createStoreWithMiddleware(reducer);

function fetchSecretSauce() {
```

```
  return fetch('https://www.google.com/search?q=secret+sauce');
}

// These are the normal action creators you have seen so far.
// The actions they return can be dispatched without any middleware.
// However, they only express “facts” and not the “async flow”.

function makeASandwich(forPerson, secretSauce) {
  return {
    type: 'MAKE_SANDWICH',
    forPerson,
    secretSauce
  };
}

function apologize(fromPerson, toPerson, error) {
  return {
    type: 'APOLOGIZE',
    fromPerson,
    toPerson,
    error
  };
}

function withdrawMoney(amount) {
  return {
    type: 'WITHDRAW',
    amount
  };
}

// Even without middleware, you can dispatch an action:
store.dispatch(withdrawMoney(100));

// But what do you do when you need to start an asynchronous action,
// such as an API call, or a router transition?

// Meet thunks.
// A thunk is a function that returns a function.
// This is a thunk.

function makeASandwichWithSecretSauce(forPerson) {

  // Invert control!
  // Return a function that accepts `dispatch` so we can dispatch later.
  // Thunk middleware knows how to turn thunk async actions into actions.

  return function (dispatch) {
    return fetchSecretSauce().then(
      sauce => dispatch(makeASandwich(forPerson, sauce)),
      error => dispatch(apologize('The Sandwich Shop', forPerson, error))
    );
  };
}

// Thunk middleware lets me dispatch thunk async actions
```

```
  // as if they were actions!

  store.dispatch(
    makeASandwichWithSecretSauce('Me')
  );

  // It even takes care to return the thunk's return value
  // from the dispatch, so I can chain Promises as long as I return them.

  store.dispatch(
    makeASandwichWithSecretSauce('My wife')
  ).then(() => {
    console.log('Done!');
  });

  // In fact I can write action creators that dispatch
  // actions and async actions from other action creators,
  // and I can build my control flow with Promises.

  function makeSandwichesForEverybody() {
    return function (dispatch, getState) {
      if (!getState().sandwiches.isShopOpen) {

        // You don't have to return Promises, but it's a handy convention
        // so the caller can always call .then() on async dispatch result.

        return Promise.resolve();
      }

      // We can dispatch both plain object actions and other thunks,
      // which lets us compose the asynchronous actions in a single flow.

      return dispatch(
        makeASandwichWithSecretSauce('My Grandma')
      ).then(() =>
        Promise.all([
          dispatch(makeASandwichWithSecretSauce('Me')),
          dispatch(makeASandwichWithSecretSauce('My wife'))
        ])
      ).then(() =>
        dispatch(makeASandwichWithSecretSauce('Our kids'))
      ).then(() =>
        dispatch(getState().myMoney > 42 ?
          withdrawMoney(42) :
          apologize('Me', 'The Sandwich Shop')
        )
      );
    };
  }

  // This is very useful for server side rendering, because I can wait
  // until data is available, then synchronously render the app.

  store.dispatch(
    makeSandwichesForEverybody()
  ).then(() =>
```

```
    response.send(React.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(
    SandwichShop,
    state => ({
      sandwiches: state.sandwiches
    })
  );
```

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 createStoreWithMiddleware = applyMiddleware(...middleware)(createStore);
```

  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 poweful 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.

`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 action creators.

2. `dispatch` (*Function*): A `dispatch` function available on the `Store` instance.

Returns

(*Function* or *Object*): An object mimicking the original object, but with each function immediately dispatching the action returned by the corresponding action creator. If you passed a function as `actionCreators`, the return value will also be a single function.

Example

`TodoActionCreators.js`

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

```
    text
  };
}

export function removeTodo(id) {
  return {
    type: 'REMOVE_TODO',
    id
  };
}
```

SomeComponent.js

```
import { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';

import * as TodoActionCreators from './TodoActionCreators';
console.log(TodoActionCreators);
// {
//   addTodo: Function,
//   removeTodo: Function
// }

class TodoListContainer extends Component {
  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(
  TodoListContainer,
  state => ({ todos: state.todos })
)
```

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.

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 * as reducers from '../reducers/index';

let reducer = combineReducers(reducers);
let middleware = [thunk];

let finalCreateStore;

// In production, we want to use just the middleware.
// In development, we want to use some store enhancers from redux-devtools.
// UglifyJS will eliminate the dead code depending on the build environment.

if (process.env.NODE_ENV === 'production') {
  finalCreateStore = applyMiddleware(...middleware)(createStore);
} else {
  finalCreateStore = compose(
    applyMiddleware(...middleware),
    require('redux-devtools').devTools(),
```

```
      require('redux-devtools').persistState(
        window.location.href.match(/[?&]debug_session=([^&]+)\b/)
      )
    )(createStore);

    // Same code without the `compose` helper:
    //
    // finalCreateStore = applyMiddleware(middleware)(
    //   require('redux-devtools').devTools()(
    //     require('redux-devtools').persistState(window.location.href.match(/[?&]debug
    //   )
    // )(createStore);
  }

  let store = finalCreateStore(reducer);
```

Tips

- All `compose` does is let you write deeply nested function transformations without the rightward drift of the code. Don’t give it too much credit!

Change Log

This project adheres to Semantic Versioning.
Every release, along with the migration instructions, is documented on the Github Releases page.

Patrons

The work on Redux was funded by the community.
Meet some of the outstanding companies and individuals that made it possible:

- Webflow
- Chess iX
- Herman J. Radtke III
- Ken Wheeler
- Chung Yen Li
- Sunil Pai
- Charlie Cheever
- Eugene G
- Matt Apperson
- Jed Watson
- Sasha Aickin