EVERYTHING YOU NEED
TO BUILD REAL PROJECTS
WITH REDUX

THE COMPLETE REDUX BOOK

BORIS DINKEVICH
ILYA GELMAN

The Complete Redux Book

Everything you need to build real projects with Redux

Ilya Gelman and Boris Dinkevich

This book is for sale at http://leanpub.com/redux-book

This version was published on 2018-04-19

ISBN 978-965-92642-0-9

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2018 Ilya Gelman and Boris Dinkevich

Tweet This Book!

Please help Ilya Gelman and Boris Dinkevich by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Time to learn Redux!

The suggested hashtag for this book is #ReduxBook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ReduxBook

Contents

Preface . . . . **i**
- Should I Read This Book? i
- How to Read This Book i
- Code Repository ii
- Acknowledgements ii

Part 1. Introduction to Redux 1

Chapter 1. Core Concepts of Flux and Redux **2**
- What Is Flux? 2
- Redux and Flux 4
- Redux Terminology 6
- General Concepts 9
- Redux and React/Angular 12
- Basic Redux Implementation 13
- Summary 19

Chapter 2. Example Redux Application **20**
- Starter Project 20
- Example Application 22
- A Closer Look at Reducers 33
- Handling Typos and Duplicates 36
- A Simple UI 36
- Logging 40
- Getting Data from the Server 41
- Summary 44

Part 2. Redux in Depth 45

Chapter 3. The Store **46**
- Creating a Store 46

Decorating the Store 51
Summary 58

Chapter 4. Actions and Action Creators **59**
Passing Parameters to Actions 60
Action Creators 61
Flux Standard Actions 64
String Constants 65
Full Action Creators Example 67
Testing Action Creators 68
redux-thunk 69
redux-actions 74
Summary 78

Chapter 5. Reducers **79**
Reducers in Practice 79
Avoiding Mutations 86
Ensuring Immutability 92
Using Immer for Temporary Mutations 93
Higher-Order Reducers 96
Summary 97

Chapter 6. Middleware **98**
Understanding next() 99
Our First Middleware 99
Async Actions 101
Using Middleware for Flow Control 105
Other Action Types 107
Parameter-Based Middleware 108
The Difference Between next() and dispatch() 109
How Are Middleware Used? 109
Summary 109

Part 3. Redux in the Real World **110**

Chapter 7. State Management **111**
Reducer Nesting and Coupling 112
State as a Database 114
Keeping a Normalized State 118
Persisting State 123
Real-World State 126
Summary 128

Chapter 8. Server Communication **129**
Asynchronous Action Creators 130
API Middleware 131
Moving Code from Action Creators 132
Error Handling 136
Authentication 140
Handling Other HTTP Methods 140
Implementing a Loading Indicator (Spinner) 141
Transforming Data 144
Chaining Network Requests 148
Summary 149

Chapter 9. Managing Side Effects **150**
Side Effects in Action Creators 151
Side Effects in Middleware 152
Other Solutions 152
Messaging Patterns 153
Summary 157

Chapter 10. WebSockets **158**
Basic Architecture 158
Connecting to Redux 159
Implementation 159
Complete WebSocket Middleware Code 165
Authentication 166
Summary 168

Chapter 11. Testing **169**
Directory Organization 169
Testing Action Creators 170
Async Action Creators 174
Testing Reducers 180
Testing Middleware 188
Integration Tests 197
Summary 199

The Evolution of Redux **200**
Redux Roadmap 200
Growth of the Redux Ecosystem 201

Further Reading **202**
Courses and Tutorials 202
Useful Libraries 202

Resource Repositories 203

Closing Words **204**
Improving the Book 204
What's Next 204

Preface

Should I Read This Book?

There are many tutorials and blog posts about Redux on the Internet. The library also has great official documentation. This book isn't supposed to be either a tutorial or documentation. The goal is to provide a methodical explanation of Redux's core concepts and how those can be extended and used in large and complex Redux applications.

As a front-end consultancy, at 500Tech we help dozens of companies build great projects using Redux. Many projects face the same problems and pose the same questions. How should we structure the application? What is the best way to implement server communication? What is the best solution for form validation? Where should we handle side effects? How will Redux benefit our applications?

This book is intended to serve as a companion for developers using Redux on a daily basis. It aims to give answers to many of the above questions and provide solutions to the most common problems in real-world applications. It can be used to learn Redux from the ground up, to better understand the structure of a large application, and as a reference during development.

The book is structured in a way that doesn't force the reader to read it start to finish, but rather allows you to skip parts and come back to them when you face the problem at hand or have free time to deepen your knowledge. We love Redux, and we have tried to share our excitement about it in this book. We hope that you will find it useful.

How to Read This Book

While Redux in itself is a small library, the underlying concepts and the ecosystem around it are large and complex. In this book we cover the core and common concepts and methods a developer needs to work with Redux on both small and large-scale applications.

The book is separated into three parts. In the first part you will learn the basics of Redux. Chapter 1 covers the core concepts behind Redux, introducing the different "actors" and how it is built. In Chapter 2, we build an example project step by step; here, you will learn how to use Redux in a project.

The second part is a deep dive into Redux usage. It is separated into chapters by Redux entity types: actions, middleware, reducers, and the store and store enhancers. The chapters in this section include advanced explanations of Redux internals and how to properly use them in complex scenarios, and they are a must-read for anyone considering building large applications with Redux.

The third part of the book contains examples and use cases from real applications. Redux has a great ecosystem, and there are a lot of tips, tricks, and libraries that can be applied to many projects of different scale. We provide you with solutions for common problems including server communication, state management, testing, and more.

It is highly recommended that you start by reading the first part as a whole. Even if you already have knowledge of Redux and have used it, the opening chapters will clarify all the underlying concepts and lay down the groundwork for the code we will use throughout the second and third parts of the book.

No matter who you are—an explorer learning about Redux for fun, or a hard-working professional who needs to solve real problems fast—this book will provide new ideas and insights to help you on your path.

Code Repository

The code samples from this book are available in the book's repository on GitHub[1] and should work in all modern browsers.

Acknowledgements

Writing a technical book on a popular JavaScript library nowadays isn't a simple task. New techniques, best practices, and opinions keep popping up every other week. Combined with a day job and a family, it's even harder. The only way we could succeed is with help from other awesome people along the way.

To Redux's creators, Dan Abramov and Andrew Clark, as well as to the many contributors to Redux and its ecosystem, thank you for improving data management and making this book relevant.

To our technical copyeditor, Rachel Head[2], thank you so much for fixing our English and making this book more understandable.

To all our colleagues at 500Tech[3], thanks for being awesome and making us feel good every day.

And obviously thank you, our dear readers, for deciding to spend your time and money on reading this book. We hope you enjoy reading it as much as we did writing it.

—Boris & Ilya

[1] https://github.com/redux-book

[2] https://fr.linkedin.com/in/rachel-head-a45258a2

[3] http://500tech.com

Part 1. Introduction to Redux

Chapter 1. Core Concepts of Flux and Redux

Penicillin, x-rays, and the pacemaker are famous examples of unintended discoveries. Redux, in a similar way, wasn't meant to become a library, but turned out to be a great Flux implementation. In May 2015, one of its authors, Dan Abramov[4], submitted a talk to the ReactEurope conference about "hot reloading and time travel." He admits he had no idea how to implement time travel at that point. With help from Andrew Clark[5] and inspired by some elegant ideas from the Elm[6] language, Dan eventually came up with a very nice architecture. When people started catching on to it, he decided to market it as a library.

In less than half a year, that small (only 2 KB) library became the go-to framework for React developers, as its tiny size, easy-to-read code, and very simple yet neat ideas were much easier to get to grips with than competing Flux implementations. In fact, Redux is not exactly a Flux library, though it evolved from the ideas behind Facebook's Flux architecture. The official definition of Redux is *a predictable state container for JavaScript applications*. This means that you store all of your application state in one place and can know what the state is at any given point in time.

What Is Flux?

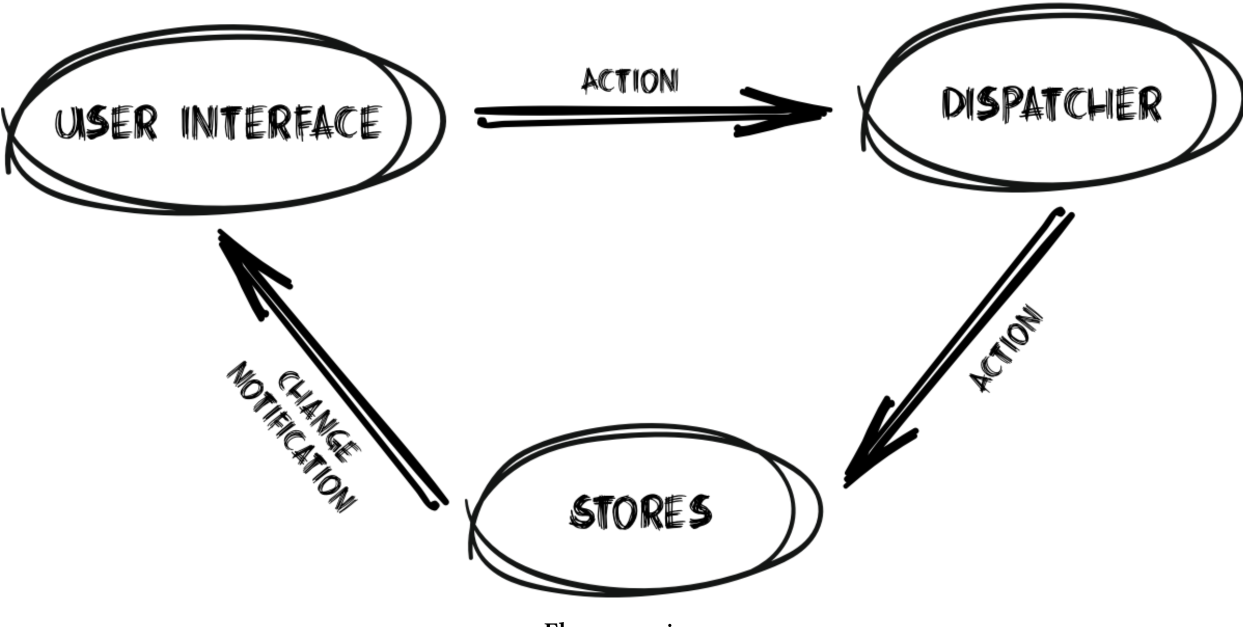
Before diving into Redux, we should get familiar with its base and predecessor, the Flux architecture. Flux is a generic architecture or pattern, rather than a specific implementation. Its ideas were first introduced publicly[7] by Bill Fisher and Jing Chen at the Facebook F8 conference in April 2014. Flux was touted as redefining the previous ideas of MVC (Model–View–Controller) and MVVM (Model–View–ViewModel) patterns and two-way data binding introduced by other frameworks by suggesting a new flow of events on the front end, called the *unidirectional data flow*.

In Flux, events are managed one at a time in a circular flow with a number of actors: dispatcher, stores, and actions. An *action* is a structure describing any change in the system: mouse clicks, timeout events, network requests, and so on. Actions are sent to a *dispatcher*, a single point in the system where anyone can submit an action for handling. The application state is then maintained in *stores* that hold parts of the application state and react to commands from the dispatcher.

[4] http://survivejs.com/blog/redux-interview/
[5] https://twitter.com/acdlite
[6] http://elm-lang.org
[7] https://www.youtube.com/watch?v=i__969noyAM

Here is the simplest Flux flow:

1. Stores subscribe to a subset of actions.
2. An action is sent to the dispatcher.
3. The dispatcher notifies subscribed stores of the action.
4. Stores update their state based on the action.
5. The view updates according to the new state in the stores.
6. The next action can then be processed.



Flux overview

This flow ensures that it's easy to reason about how actions flow in the system, what will cause the state to change, and how it will change.

Consider an example from a jQuery or AngularJS (prior to version 2) application. A click on a button can cause multiple callbacks to be executed, each updating different parts of the system, which might in turn trigger updates in other places. In this scenario it is virtually impossible for the developer of a large application to know how a single event might modify the application's state, and in what order the changes will occur.

In Flux, the click event will generate a single action that will mutate the store and then the view. Any actions created by the store or other components during this process will be queued and executed only after the first action is done and the view is updated.

Flux as a Library

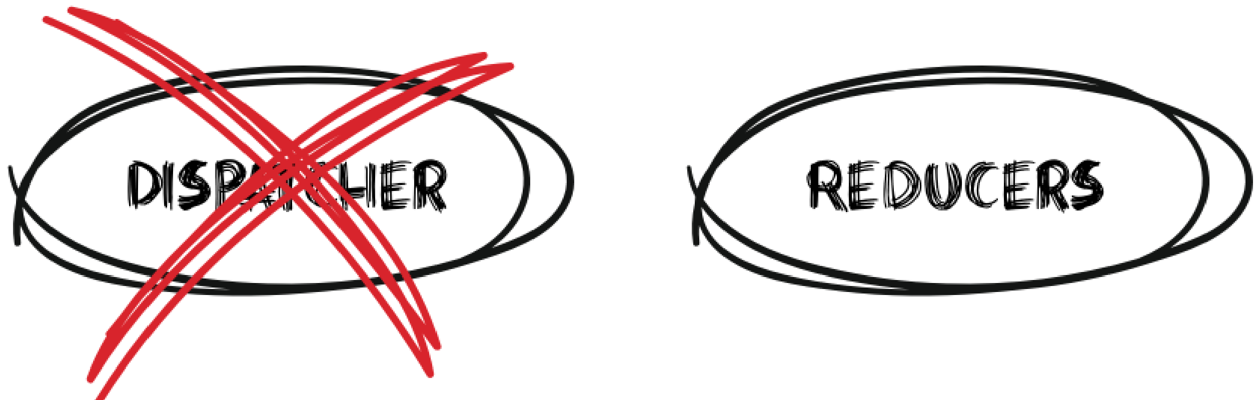
Facebook's developers did not initially open-source their implementation of Flux, but rather released only parts of it, like the dispatcher. The concept of the Flux flow was immediately taken up by the JavaScript community and caused a lot of different open-source implementations to be built and released in quick succession.

Some of the new Flux libraries followed the basic Flux pattern very closely, while others differed significantly from the original patterns. For example, some moved to having multiple dispatchers or introduced dependencies between stores.

Dmitri Voronianski has a good comparison of various Flux implementations on GitHub[8].

Redux and Flux

While Redux derives from Flux concepts, there are a few distinctions between the two architectures. In contrast to Flux, Redux only has a single store that holds no logic by itself. Actions are dispatched and handled directly by the store, eliminating the need for a standalone dispatcher. In turn, the store passes the actions to state-changing functions called *reducers*, a new type of actor added by Redux.



Dispatcher out, reducers in

[8] https://github.com/voronianski/flux-comparison

Application Data

To better understand Redux, let's imagine an application that helps us manage a recipe book. All the data for our recipe book application, consisting of a list of recipes and their details, will be stored in a single large object: the *store*. It is best to compare this pattern to having a database on the front end of your application, much like the database on the server side, that might hold multiple tables (or collections) for each data object.

While many other frameworks divide the data between different services and areas, in Redux we keep all our data in a central repository accessible by all parts of the UI.

Changing the Data

Since all of our data is sitting in a single JavaScript object, there has to be a way to modify it in a clear and consistent fashion. But allowing various places in our code to directly access and modify our central repository will make it very hard to track changes and to correctly update the UI.

In Redux, all changes to the store are initiated by sending an *action*, a plain JavaScript object containing all the information describing the required change. The action is sent (dispatched) to our store, which in turn calculates the new state according to the action.

Since the store is a generic implementation, in Redux we use another concept—*reducers*—to calculate what our current state will look like once an action is applied to it. For example, adding a new recipe changes the old state, say with an array of three recipes, to a new one with an array of four.

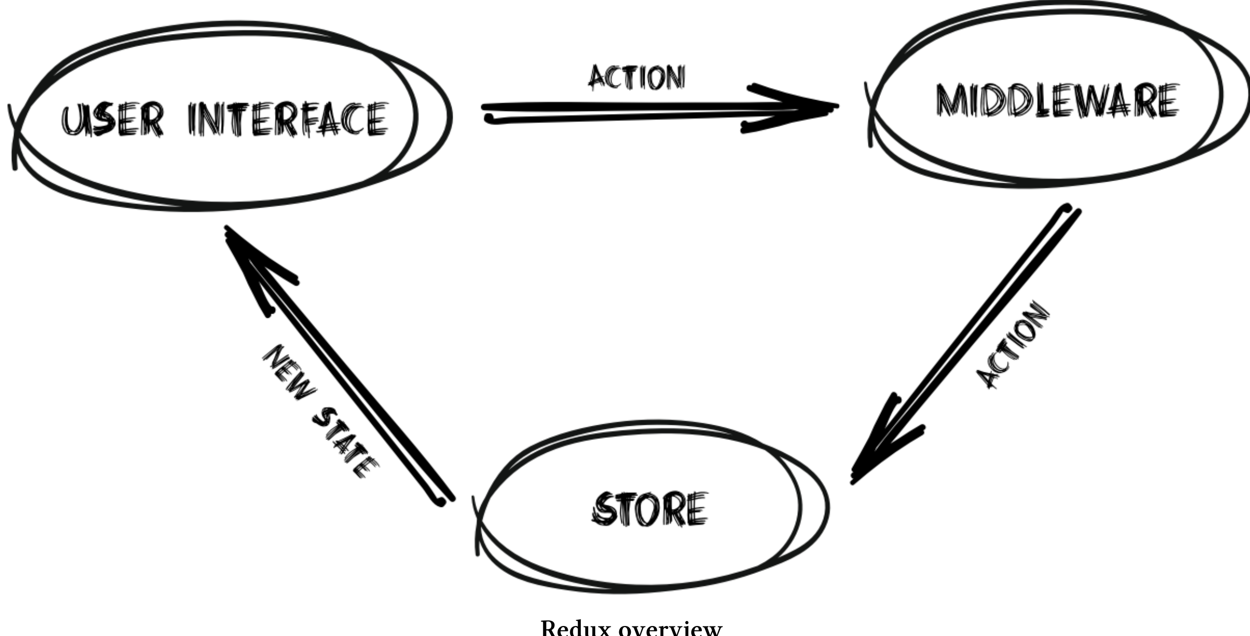
Updating the UI

Each UI framework using Redux (React, Angular, etc.) is responsible for subscribing to the store to listen to its "store updated" events and updating the UI accordingly.

The core concept of Redux is that our UI always reflects the state of the application in the store. Sending an action will cause our store to use our reducers to calculate a new state and notify the UI layer to update the UI accordingly.

Advanced Flows

There are other parts of Redux that make the application easier to build and manage, like the *middleware*. Every action gets passed through a pipeline of middleware. Unlike reducers, middleware can modify, stop, or add more actions. Examples might include logging middleware, authorization middleware that checks if the user has the necessary permissions to run the action, or API middleware that sends something to the server.



Redux overview

Redux Terminology

Before going any further, let's make these concepts more concrete with some examples.

Actions and Action Creators

The only way for an application to change its state is by processing *actions*. In most cases, actions in Redux are nothing more than plain JavaScript objects passed to the store that hold all the information needed for the store to be able to modify the state:

Example of an action object

```
{
  type: 'INCREMENT',
  payload: {
    counterId: 'main',
    amount: -10
  }
}
```

Since these objects might have some logic and can be used in multiple places in an application, they are commonly wrapped in a function that can generate the objects based on a parameter:

A function that creates an action object

```
function incrementAction(counterId, amount) {
  return {
    type: 'INCREMENT',
    payload: {
      counterId,
      amount
    }
  };
};
```

As these functions create action objects, they are aptly named *action creators*.

Reducers

Once an action is sent to the store, the store needs to figure out how to change the state accordingly. To do so, it calls a function, passing it the current state and the received action:

A function that calculates the next state

```
function calculateNextState(currentState, action) {
  ...
  return nextState;
}
```

This function is called a *reducer*. In real Redux applications, there will be one *root reducer* function that will call additional reducer functions to calculate the nested state:

A simple reducer implementation

```
function rootReducer(state, action) {
  switch (action.type) {

    case 'INCREMENT':
      return Object.assign({}, state, {
        counter: state.counter + action.payload.amount
      });

    default:
      return state;
  }
}
```

This sample reducer copies the original state into a new JavaScript object and overwrites the `counter` key with an updated value. The reducer does not change the original `state` parameter passed to it, keeping it immutable.

Reducers never modify the original state; they always create a new copy with the needed modifications.

Middleware

Middleware is a more advanced feature of Redux and will be discussed in detail in later chapters. Middleware act like interceptors for actions before they reach the store: they can modify the actions, create more actions, suppress actions, and much more. Since middleware have access to the actions, the `dispatch()` function, and the store, they are the most versatile and powerful entities in Redux.

Store

Unlike many other Flux implementations, Redux has a single store that holds the application information but no user logic. The role of the store is to receive actions, pass them through all the registered middleware, and then use reducers to calculate a new state and save it.

When it receives an action that causes a change to the state, the store will notify all registered listeners that a change to the state has been made. This will allow various parts of the system, like the UI, to update themselves according to the new state.

General Concepts

Redux is all about functional programming and pure functions. Understanding these concepts is crucial to understanding the underlying principles of Redux.

Functional programming has become a trendy topic in the web development domain lately, but it was actually invented around the 1950s. The paradigm centers around avoiding changing state and mutable data—in other words, making your code predictable and free of side effects.

JavaScript allows you to write code in a functional style, as it treats functions as first-class objects. This means you can store functions in variables, pass them as arguments to other functions, and return them as values of other functions. But since JavaScript was not designed to be a functional programming language per se, there are some caveats that you will need to keep in mind. In order to get started with Redux, you need to understand pure functions and mutation.

Pure and Impure Functions

A *pure function* returns values by using only its arguments: it uses no additional data and changes no data structures, touches no storage, and emits no external events (like network calls). This means that you can be completely sure that every time you call the function with the same arguments, you will always get the same result. Here are some examples of pure functions:

Examples of pure functions

```
function square(x) {
  return x * x;
}

Math.sin(y);

arr.map((item) => item.id);
```

If a function uses any variables not passed in as arguments or creates side effects, the function is *impure*. When a function depends on variables or functions outside of its lexical scope, you can never be sure that the function will behave the same every time it's called. For example, the following are impure functions:

Examples of impure functions

```
function getUser(userId) {
  return UsersModel.fetch(userId).then((result) => result);
}

Math.random();

arr.map((item) => calculate(item));
```

Mutating Objects

Another important concept that often causes headaches for developers starting to work with Redux is *immutability*. JavaScript has limited tooling for managing immutable objects, and we are often required to use external libraries.

Immutability means that something can't be changed, guaranteeing developers that if you create an object, it will have the same properties and values forever. For example, let's declare a simple object as a constant:

An object defined as a constant in JavaScript

```
const colors = {
  red: '#FF0000',
  green: '#00FF00',
  blue: '#0000FF'
};
```

Even though the `colors` object is a constant, we can still change its content, as `const` will only check if the *reference* to the object has changed:

JavaScript allows changes in objects defined as constants

```
colors = {};
console.log(colors);

colors.red = '#FFFFFF';
console.log(colors.red);
```

Try writing this in the developer console. You will see that you can't reassign an empty object to `colors`, but you can change its internal value.

To make the `colors` object appear immutable, we can use the `Object.freeze()` method:

Making a plain object immutable

```
Object.freeze(colors);

colors.red = '#000000';

console.log(colors.red);
```

The value of the `red` property will now be `'#FFFFFF'`. If you thought that the value should be `'#FF0000'`, you missed that we changed the `red` property before we froze the object. This is a good example of how easy it is to overlook this kind of thing in real applications.

Here, once we used `Object.freeze()`, the `colors` object became immutable. In practice things are often more complicated, though. JavaScript does not provide good native ways to make data structures fully immutable. For example, `Object.freeze()` won't freeze nested objects:

Object.freeze() does not freeze nested objects

```
const orders = {
  bread: {
    price: 10
  },
  milk: {
    price: 20
  }
};

Object.freeze(orders);

orders.milk.price -= 15;

console.log(orders.milk.price);
```

To work around the nature of our beloved language, we have to use third-party libraries like `deep-freeze`[9], `seamless-immutable`[10], or Immutable.js[11].

Redux and React/Angular

Redux started out as a companion to React[12], but it has started to gather a major following with other frameworks, like Angular[13]. At its base Redux is fully framework-agnostic, and it can easily be used with any JavaScript framework to handle state and changes.

Redux—or its concepts—is used on the server side too, with implementations in Java, Python, and more. There are even implementations for other platforms, such as ReSwift[14] for iOS.

The connection to different frameworks is done with the help of third-party libraries that provide a set of convenience functions for each framework in order to seamlessly connect to Redux. Since this book aims to be framework-agnostic we won't be covering the various connectors, but most are relatively simple and provide ample documentation on setting up the connections.

[9] https://github.com/substack/deep-freeze
[10] https://github.com/rtfeldman/seamless-immutable
[11] https://facebook.github.io/immutable-js/
[12] https://github.com/reactjs/react-redux
[13] https://github.com/angular-redux/ng-redux
[14] https://github.com/ReSwift/ReSwift

Basic Redux Implementation

People love Redux because of its simplicity. In fact, it is so simple that we can implement most of it in a handful of lines of code. Thus, unlike with other frameworks, where the only way to learn is to study the API, here we can start by implementing Redux ourselves.

The basic premise behind Redux is the idea that all the application state is saved in one place, the store. To use this idea in applications we will need to find a way to:

1. Modify the state as a result of events (user-generated or from the server).
2. Monitor state changes so we can update the UI.

The first part can be split into two blocks of functionality:

1. Notify the store that an action has happened.
2. Help the store figure out how to modify the state according to our application's logic.

Using this structure, let's build a simple application that will implement a counter. Our application will use pure JavaScript and HTML and require no additional libraries in any modern browser. We are going to have two buttons that will allow us to increment and decrement a simple counter, and a place where we can see the current counter value:

index.html

```
<div>
  Counter:
  <span id='counter'></span>
</div>

<button id='inc'>Increment</button>
<button id='dec'>Decrement</button>
```

Our application state will simply hold the counter value:

A simple state holding a counter

```
let state = {
  counter: 3
};
```

To make our demo functional, let's create a click handler for each button that will use the `dispatch()` function to notify our store that an action needs to be performed:

```
dispatch(action);
```

Connecting click events to dispatch()

```
// Listen to click events
document.querySelector('#inc').onclick = () => dispatch('INC');
document.querySelector('#dec').onclick = () => dispatch('DEC');
```

We will come back to its implementation later in this chapter. Also, let's define a function that will update the counter's value in the HTML based on application state received as an argument:

Code to update the counter in the DOM

```
// Function to update view (this might be React or Angular in a real app)
function updateView() {
  document.querySelector('#counter').innerText = state.counter;
}
```

Since we want our view to represent the current application state, we need it to be updated every time the state (and the counter) changes. For that, we will use the `subscribe()` function, which we will also implement a bit later. The role of the function will be to call our callback every time anything in the state changes:

```
subscribe(updateView);
```

We have now created a basic application structure with a simple state, implemented a function that will be responsible for updating the HTML based on the state, and defined two "magic" functions—`dispatch()` and `subscribe()`—to dispatch actions and subscribe to changes in the state. But there is still one thing missing. How will our mini-Redux know how to handle the events and change the application state?

For this, we define an additional function. On each action dispatched, Redux will call our function, passing it the current state and the action. To be compliant with Redux's terminology, we will call the function a *reducer*. The job of the reducer will be to understand the action and, based on it, create a new state.

In our simple example our state will hold a counter, and its value will be incremented or decremented based on the action:

Simple reducer for INC and DEC actions

```
// Our mutation (reducer) function creates a new state based on the action passed
function reducer(state, action) {
  switch (action) {
    case 'INC':
      return Object.assign({}, state, { counter: state.counter + 1 });
    case 'DEC':
      return Object.assign({}, state, { counter: state.counter - 1 });
    default:
      return state;
  }
}
```

An important thing to remember is that reducers must always return a new, modified copy of the state. They shouldn't mutate the existing state, like in this example:

Incorrect way to change state

```
// This is wrong!
state.counter = state.counter + 1;
```

Later in the book you will learn how you can avoid mutations in JavaScript with and without the help of external libraries.

Now it's time to implement the actual change of the state. Since we are building a generic framework, we will not include the code to increment/decrement the counter (as it is application-specific) but rather will call a function that we expect the user to supply, called `reducer()`. This is the reducer we mentioned before.

The `dispatch()` function calls the `reducer()` implemented by the application creator, passing it both the current state and the action it received. This information should be enough for the `reducer()` function to calculate a new state. We then check if the new state differs from the old one, and if it does, we replace the old state and notify all the listeners of the change:

dispatch() implementation

```
let state = null;

function dispatch(action) {
  const newState = reducer(state, action);

  if (newState !== state) {
    state = newState;

    listeners.forEach(listener => listener());
  }
}
```

Again, it is very important to note that we expect a reducer to create a *new state* and not just modify the existing one. We will be using a simple comparison by reference to check whether the state has changed.

One remaining task is to notify our view of the state change. In our example we only have a single listener, but we already can implement full listener support by allowing multiple callbacks to register for the "state change" event. We will implement this by keeping a list of all the registered callbacks:

subscribe() implementation

```
const listeners = [];

function subscribe(callback) {
  listeners.push(callback);
}
```

This might surprise you, but we have just implemented the major part of the Redux framework. The real code[15] isn't much longer, and we highly recommended that you take half an hour to read it.

[15] https://github.com/reactjs/redux/tree/master/src

Using the Real Redux

To complete our example, let's switch to the real Redux library and see how similar the solution remains. First we'll add the Redux library, for now using unpkg:

Adding Redux to a project

```
<script src="https://unpkg.com/redux/dist/redux.js" />
```

We then change our previous state definition to be a constant that only defines the initial value of the state:

The initial state

```
const initialState = {
  counter: 3
};
```

Now we can use it to create a Redux store:

Creating a Redux store

```
const store = Redux.createStore(reducer, initialState);
```

As you can see, we are using our reducer from before. The only change that needs to be made to the reducer is in the `switch` statement. Instead of using just the action:

Previous reducer condition

```
switch (action)
```

we include the mandatory `type` property, which indicates the type of action being performed:

New reducer condition

```
switch (action.type)
```

The Redux store will also give us all the features we implemented ourselves before, like `subscribe()` and `dispatch()`. Thus, we can safely delete these methods.

To subscribe to store changes, we simply call the `subscribe()` method of the store:

Subscribing to store updates

```
store.subscribe(updateView);
```

Since `subscribe()` does not pass the state to the callback, we need to access it via `store.getState()`:

Updating view by reading the state from the store

```
// Function to update view (this might be React or Angular in a real app)
function updateView() {
  document.querySelector('#counter').innerText = store.getState().counter;
}

store.subscribe(updateView);
```

The last change is in the `dispatch()` method. As mentioned previously, our actions now need to have the `type` property. Thus, instead of sending the string `'INC'` as the action, we now need to send `{ type: 'INC' }`.

The Complete Example

HTML

```
<script src="https://unpkg.com/redux/dist/redux.js"></script>

<div>
  Counter:
  <span id='counter'> </span>
</div>

<button id='inc'>Increment</button>
<button id='dec'>Decrement</button>
```

JavaScript

```
// Our mutation (reducer) function creates
// a _new_ state based on the action passed
function reducer(state, action) {
  switch (action.type) {
    case 'INC':
      return Object.assign({}, state, { counter: state.counter + 1 });
    case 'DEC':
      return Object.assign({}, state, { counter: state.counter - 1 });
    default:
      return state;
  }
}

const initialState = {
  counter: 3
};

const store = Redux.createStore(reducer, initialState);

// Function to update view (this might be React or Angular in a real app)
function updateView() {
  document.querySelector('#counter').innerText = store.getState().counter;
}

store.subscribe(updateView);

// Update view for the first time
updateView();

// Listen to click events
document.getElementById('inc').onclick = () => store.dispatch({ type: 'INC' });
document.getElementById('dec').onclick = () => store.dispatch({ type: 'DEC' });
```

Summary

In this chapter we briefly covered the history of Redux and Flux, and how Redux works at its core. We also discussed basic functional programming principles, such as pure functions and immutability. As they are very important for our real-world applications, we will talk about these concepts more later in the book. In the next chapter we are going to demonstrate how to actually work with Redux by building a simple recipe book application.

Chapter 2. Example Redux Application

In the previous chapter you learned what Redux is and how it works. In this chapter you will learn how to use it for a simple project. It is highly recommended that you follow along with and fully understand this chapter and its code before moving on to more advanced topics.

Starter Project

Modern client-side applications often require use of some so-called "boilerplate" in order to make development easier. The boilerplate may include things such as directory structure, code transformation tools like SCSS and ES2018 compilers, testing infrastructure, and production pipeline tools for tasks such as minification, compression, and concatenation.

To ease the chore of setting up a new project, all the popular frameworks have optional command-line tools to help developers quickly bootstrap their projects with all the required settings, such as Create React App[16], Vue CLI[17], or the heavier Angular CLI[18] that offers many more features than just project initiation.

Besides the official tools, the open-source community has created dozens of different starter projects. The larger ones, like `react-boilerplate`[19] and `react-redux-starter-kit`[20], consist of over a hundred files.

For our purposes we will use a bare-bones boilerplate, just enough to cover all the concepts explained in this book. As our project will be pure Redux, it will require no React or related libraries, and we will use the most minimal Webpack[21] configuration; just sufficient to enable use of ES2015 modules and live page reloading when we make code changes.

[16]https://github.com/facebookincubator/create-react-app
[17]https://github.com/vuejs/vue-cli
[18]https://github.com/angular/angular-cli
[19]https://github.com/react-boilerplate/react-boilerplate
[20]https://github.com/davezuko/react-redux-starter-kit
[21]https://webpack.js.org/

To start things off, we'll clone the starter project[22], install the needed packages, and verify that our environment is ready:

Setting up the starter project

```
git clone git@github.com:redux-book/starter.git
cd starter
npm install
npm start
```

If everything went smoothly, you should be able to access *http://localhost:8080* and see a page showing "A simple Redux starter" and a running counter. If you open the JavaScript console in the developer tools, you should also see "Redux started" in the console. The project is ready! Time to open the code editor and go over the four files currently making up the project.

Skeleton Overview

Let's take a quick look at all the files that are included within the starter project. The main file we will be working with is *app.js*, but it is important to get familiar with the directory structure before you start working on any project. The project files are:

package.json
: While the majority of the fields in this file are irrelevant at this point, it is important to note two sections, `devDependencies` and `dependencies`. The former contains a list of all the tools needed to build the project. It currently includes only Webpack-related packages, just to enable us to run our app. The `dependencies` section lists all the packages we will bundle with our application. It includes only the `redux` library itself, and `jquery` to make the DOM manipulation code look nicer.

webpack.config.js
: This is the main Webpack configuration file. This settings file instructs Webpack how to chain transpilation tools and how to build packages, and usually holds most of the configuration of the project's tooling. In our simple project there is only one settings file (larger projects might have more granular files for testing, development, and production). Our *webpack.config.js* file defines *app.js* as the main file, which acts as an entry point to our application and the output destination for the bundled project.

index.html, app.js
: Single-page applications, unlike their server-rendered cousins, have a single entry point. In our project every part and page of the application will be rendered starting from *index.html*, and all the JavaScript-related startup code will be in *app.js*.

[22] http://github.com/redux-book/starter

Setup

To begin, let's clean up the starter project so you have a clean slate to follow along with the instructions in this chapter.

First, remove all content from the *app.js* file. Then change the *index.html* file to the simple version below, removing all unnecessary code from the `body` section:

index.html

```html
<!DOCTYPE html>
<html>
  <head lang="en">
    <meta charset="UTF-8">
    <title>The Complete Redux Book - Example Application</title>
  </head>
  <body>
    <div id="app"></div>
    <script type="text/javascript" src="app.js"></script>
  </body>
</html>
```

The only `div` will be our *mounting point* to the single-page application. This is where we will programmatically initiate and update our UI using jQuery.

You're ready! Once you run `npm start`, the application in the browser should update automatically when you make changes to the code. The final application is available for your reference on GitHub in the full example repository[23].

Example Application

To illustrate how to use different parts of Redux, we will be building a recipe book application. It will allow adding recipes and ingredients for each recipe, and will fetch an initial recipe list from a remote server. In accordance with Redux principles, the application will keep all its state—data and UI—in the global store.

[23] https://github.com/redux-book/chapter2

The first step with any Redux-based app is to plan how data will be arranged in the store. Our recipe object will start out holding only the recipe's name. We will add more fields later, as needed. To store the current list, we can use a regular array:

Naive state structure

```
recipes = [
  { name: 'Omelette' },
  ...
];
```

Ingredients for each recipe will contain a `name` and a `quantity`. Connecting them to the state will be a bigger challenge. There are three general approaches to make this connection.

The *nested objects* approach is to hold the ingredients as an array inside a recipe itself:

Nested objects state

```
state = {
  recipes: [
    {
      name: 'Omelette',
      ingredients: [
        {
          name: 'Egg',
          quantity: 2
        }
      ]
    },
    ...
  ]
};
```

The *nested reference* approach is to store the recipe ingredient information directly in the state and hold an array of recipe ingredient IDs in each recipe:

Nested reference state

```
state = {
  recipes: [
    {
      name: 'Omelette',
      ingredients: [2, 3]
    }
  ],
  ingredients: {
    2: { name: 'Egg', quantity: 2 },
    3: { name: 'Milk', quantity: 1 }
  }
};
```

The *separate object* approach is to store the ingredients as a standalone array in the state, and put the ID of the recipe the array is connected to inside of it:

Separate objects state

```
state = {
  recipes: [
    {
      id: 10,
      name: 'Omelette'
    }
  ],
  ingredients: [
    {
      recipe_id: 10,
      name: 'Egg',
      quantity: 2
    },
    {
      recipe_id: 10,
      name: 'Milk',
      quantity: 1
    }
  ]
};
```

While all the approaches have their upsides and downsides, you will quickly discover that in Redux, keeping the structure as flat and normalized as possible (as in the second and third examples shown here) makes the code cleaner and simpler. The state's structure implies the use of two separate reducers for recipes and ingredients. We can process both independently.

The biggest difference between the second and third options is how the link is made (what holds the ID of what). In the second example, adding an ingredient will require an update in two different parts of the state—in both the `recipes` and `ingredients` subtrees—while in the third approach, we can always update only one part of the tree. In our example we will use this method.

The subject of state management is covered in detail in the State Management chapter in Part 3.

Store Setup

We will start by creating the store. In Redux there is only one store, which is created and initialized by the `createStore()` method. Let's open our *app.js* file and create the store:

Creating the Redux store

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

const reducer = (state, action) => state;
const store = createStore(reducer);

window.store = store;
```

The `createStore()` function can receive a number of parameters, with only one being required—the reducer. In our example, the reducer simply returns the same state regardless of the action.

To make things more interesting, we can provide a default state to the store. This is useful when learning, but the real use of this feature is mainly with server rendering, where we precalculate the state of the application on the server and then can create the store with the precalculated state on the client. We provide an initial state as follows:

Creating the store with an initial state

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

const initialState = {
  recipes: [{
    name: 'Omelette'
  }],
  ingredients: [{
    recipe: 'Omelette',
    name: 'Egg',
    quantity: 2
  }]
};

const reducer = (state, action) => state;
const store = createStore(reducer, initialState);

window.store = store;
```

In the last line we make the store globally available by defining it as a property of the global `window` object. If we go to the JavaScript console, we can now try playing with it:

Trying out the APIs in the console

```
store.getState()
// Object {recipes: Array[1], ingredients: Array[1]}

store.subscribe(() => console.log("Store changed"));

store.dispatch({ type: 'ACTION' });
// Store changed
```

As you can see, we can use the `store` object to access the current state using `getState()`, subscribe to get notifications on store changes using `subscribe()`, and send actions using `dispatch()`.

Adding Recipes

To implement adding recipes, we need to find a way to modify our store. As we learned in the previous chapter, store modifications can only be done by *reducers* in response to *actions*. This means we need to define an action structure and modify our (very lean) reducer to support it.

Actions in Redux are nothing more than plain objects that have a mandatory `type` property. We will be using strings to name our actions, with the most appropriate in this case being `'ADD_RECIPE'`. Since a recipe has a `name`, we will also add that to the action's data when dispatching:

Dispatching an action

```
store.dispatch({ type: 'ADD_RECIPE', name: 'Pancakes' });
```

Let's modify our reducer to support the new action. A simple approach might appear to be the following:

Reducer that supports ADD_RECIPE

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      state.recipes.push({ name: action.name });
  }

  return state;
};
```

While this looks correct (and works when tested in our simple example), this code violates the basic Redux principle of *state immutability*. Our reducers must never *change* the state, but only create a new version of it, with any modifications needed. Thus, our reducer code needs to be modified:

Correct way to build a reducer

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return Object.assign({}, state, {
        recipes: state.recipes.concat({ name: action.name })
      });
  }

  return state;
};
```

The `'ADD_RECIPE'` case has become more complex but works exactly as expected. We are using the `Object.assign()` method to create a new object that has all the key/value pairs from our old state, but overrides the `recipes` key with a new value.

To calculate the new list of recipes we use `concat()` instead of `push()`, as `push()` modifies the original array while `concat()` creates a new array containing the original values and the new one.

More information about the `Object.assign()` method is available in the Reducers chapter in Part 2.

Adding Ingredients

Similar to adding recipes, this step will require us to modify the reducer again to add support for adding ingredients:

Adding ADD_INGREDIENT to the reducer

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return Object.assign({}, state, {
        recipes: state.recipes.concat({ name: action.name })
      });

    case 'ADD_INGREDIENT':
      const newIngredient = {
        name: action.name,
        recipe: action.recipe,
        quantity: action.quantity
      };
      return Object.assign({}, state, {
        ingredients: state.ingredients.concat(newIngredient)
      });
   }

  return state;
};
```

Let's test the new functionality by dispatching some actions from the browser console:

Checking new reducer functionality

```
store.getState();

store.dispatch({ type: 'ADD_RECIPE', name: 'Pancakes' });

store.dispatch({
  type: 'ADD_INGREDIENT',
  recipe: 'Pancakes',
  name: 'Egg',
  quantity: 3
});

store.getState();
```

If you followed the instructions correctly, you should be able to see the new recipe and ingredient in the state.

One problem you might encounter while dispatching actions from the console to test the store is that it's hard to remember the properties that need to be passed in the action object. This, among other reasons, is why in Redux we use the concept of *action creators*: functions that create the action objects for us. For example:

A function to create the action object

```
const addIngredient = (recipe, name, quantity) => ({
  type: 'ADD_INGREDIENT', recipe, name, quantity
});

store.dispatch(addIngredient('Pancakes', 'Egg', 3));
```

This function both hides the structure of the action from the user and allows us to modify the action, setting default values for properties, performing cleanup, trimming names, and so on.

For more information on action creators, consult the Actions and Action Creators chapter in Part 2.

Structuring the Code

Having all our code in a single file is obviously a bad idea. In Redux, it's common for the directory structure to follow the names of the Redux "actors." Reducers are placed in the *reducers* directory, with the main reducer (commonly called the *root reducer*) in the *root.js* file. Action creators go in the *actions* directory, divided by the type of object or data they handle—in our case, *actions/recipes.js* and *actions/ingredients.js*.

Let's start with the actions. First we will create the *actions* directory. Inside of it we will place two files: *recipes.js* and *ingredients.js*. They will hold our action creators, logically separated by features. We will also export the action creators using ES2015 syntax so we can use them in other files:

actions/recipes.js

```
export const addRecipe = (name) => ({
  type: 'ADD_RECIPE', name
});
```

actions/ingredients.js

```
export const addIngredient = (recipe, name, quantity) => ({
  type: 'ADD_INGREDIENT', recipe, name, quantity
});
```

Now it's time for the reducers. We create a directory called *reducers* and put our only reducer in the *root.js* file within that directory, renaming it to `rootReducer`. We can export it using the default export since we will only have one reducer per file:

reducers/root.js

```
const rootReducer = (state, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return Object.assign({}, state, {
        recipes: state.recipes.concat({ name: action.name })
      });

    case 'ADD_INGREDIENT':
      const newIngredient = {
        name: action.name,
        recipe: action.recipe,
        quantity: action.quantity
      };

      return Object.assign({}, state, {
        ingredients: state.ingredients.concat(newIngredient)
      });
  }

  return state;
};

export default rootReducer;
```

Finally, since we only have a single store, we can put all its configuration code in one file, *store.js*. We will now use the new `rootReducer` we just created, and we'll export the store so we can use it later in our application:

store.js

```
import { createStore } from 'redux';
import rootReducer from './reducers/root';

const initialState = {
  recipes: [{ name: 'Omelette' }],
  ingredients: [{ recipe: 'Omelette', name: 'Egg', quantity: 2 }]
};

const store = createStore(rootReducer, initialState);

window.store = store;

export default store;
```

After making all these changes, we can use the store and action creators in our main *app.js* file. Try the following:

app.js

```
import store from './store';
import { addRecipe } from './actions/recipes';
import { addIngredient } from './actions/ingredients';

store.dispatch(addRecipe('Pancakes'));
store.dispatch(addIngredient('Pancakes', 'Egg', 3));
```

If you've made no mistakes, you should be able to see the updated store using the `store.getState()` method in your browser's console.

A Closer Look at Reducers

If you open the *reducers/root.js* file, you will find that the same reducer is now taking care of different parts of our state tree. As our application grows, more properties will be added to both the `recipes` and the `ingredients` subtrees. Since the code in both handlers is not interdependent, we can split it further into three reducers, two that are responsible for different parts of the state and one to combine them:

Multi-responsibility reducer

```
const recipesReducer = (recipes, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return recipes.concat({name: action.name});
  }

  return recipes;
};

const ingredientsReducer = (ingredients, action) => { ... }

const rootReducer = (state, action) => {
  return Object.assign({}, state, {
    recipes: recipesReducer(state.recipes, action),
    ingredients: ingredientsReducer(state.ingredients, action)
  });
};
```

Since we are using multiple reducers, let's extract them into their own files to follow the rule of one reducer per file. We can import the subreducers into the root reducer, which will in turn be used by the store:

reducers/recipes.js

```
const recipesReducer = (recipes = [], action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return recipes.concat({ name: action.name });
  }

  return recipes;
};

export default recipesReducer;
```

reducers/ingredients.js

```
const ingredientsReducer = (ingredients = [], action) => {
  switch (action.type) {
    case 'ADD_INGREDIENT':
      const newIngredient = {
        name: action.name,
        recipe: action.recipe,
        quantity: action.quantity
      };

      return ingredients.concat(newIngredient);
  }

  return ingredients;
};

export default ingredientsReducer;
```

reducers/root.js

```
import recipesReducer from './recipes';
import ingredientsReducer from './ingredients';

const rootReducer = (state, action) => {
  return Object.assign({}, state, {
    recipes: recipesReducer(state.recipes, action),
    ingredients: ingredientsReducer(state.ingredients, action)
  });
};

export default rootReducer;
```

There are three main benefits to extracting subtree management functionality into separate reducers:

1. The root reducer only creates a new state object by combining the old state and the results of each of the subreducers.
2. The recipes reducer is much simpler as it only has to handle the `recipes` part of the state.
3. All the other reducers—root, ingredients, and any future ones—don't need to know or care about the internal structure of the `recipes` subtree. Thus, changes to that part of the state tree will only require changes to the recipes reducer.

A side effect of the third point is that we can tell each reducer how to initialize its own subtree instead of relying on a big `initialState` passed as an argument to the `createStore()` function. Setting the initial state for each reducer can be done by using default parameters from ES2015:

Initial state for recipes reducer

```
const initialState = [];

const recipesReducer = (recipes = initialState, action) => { ... };
```

Go ahead and define an initial state for each subreducer, and then remove the `initialState` definition and argument from *store.js*:

store.js without initial state

```
import { createStore } from 'redux';
import rootReducer from './reducers/root';

const store = createStore(rootReducer);

window.store = store;

export default store;
```

Since combining multiple reducers is a very common pattern, Redux has a special utility function, `combineReducers()`, which does exactly what our root reducer does. Let's replace our implementation in *reducers/root.js* using this `combineReducers()` function:

Combining multiple reducers

```
import { combineReducers } from 'redux';
import recipesReducer from './recipes';
import ingredientsReducer from './ingredients';

const rootReducer = combineReducers({
  recipes: recipesReducer,
  ingredients: ingredientsReducer
});

export default rootReducer;
```

Here we created a root reducer that employs two subreducers, one sitting in the `recipes` subtree and the other in the `ingredients` subtree, each responsible for setting its own initial state.

Handling Typos and Duplicates

Before moving forward, we need to make one last change to our code to fix something that might not be an apparent problem right now. We have been using strings like `'ADD_RECIPE'` in our action creators and reducers, but never bothered to verify that they match. In large applications this often leads to errors that are very hard to debug, as a typo in the action creator will cause a reducer to ignore an action. Or, even worse, two developers might use the same string by mistake, which will lead to very strange side effects as unintended reducers will process the dispatched actions.

To fix these problems we can utilize ES2015's native `const` support, which guarantees that we cannot define the same constant twice in the same file. This will catch duplicate names at compile time, even before our code reaches the browser.

Let's create a new file, *constants/actionTypes.js*, which will hold all the action type constants in our application:

constants/actionTypes.js

```
export const ADD_RECIPE = 'ADD_RECIPE';
export const ADD_INGREDIENT = 'ADD_INGREDIENT';
```

Now in our reducers and action creators we will use the constants instead of the strings:

Using constants

```
import { ADD_RECIPE } from '../constants/actionTypes';

const recipesReducer = (recipes = [], action) => {
  switch (action.type) {
    case ADD_RECIPE:
      ...
```

Go ahead and replace strings with action types in all your reducers and actions.

A Simple UI

To give you an idea of how a basic UI can be connected to Redux, we will be using a bit of jQuery magic. Note that this example is very simple and should never be used in a real application, although it should give you a general feeling of how "real" applications connect to Redux.

Let's store our current UI in *ui.js*. The jQuery UI will create a simple view of the current recipes in the store:

ui.js

```
import $ from 'jquery';
import store from './store';

function updateUI() {
  const { recipes } = store.getState();
  const renderRecipe = (recipe) => `<li>${ recipe.name }</li>`;

  $('.recipes > ul').html(recipes.map(renderRecipe));
}

export default function loadUI() {
  $('#app').append(`
    <div class="recipes">
      <h2>Recipes:</h2>
      <ul></ul>
    </div>
  `);

  store.subscribe(updateUI);

  updateUI();
}
```

We are using jQuery's `append()` method to add a new `div` to our application container and using the `updateUI()` function to pull the `recipes` list from our state and display the recipes as a list of unordered elements.

To make our UI respond to updates, we can simply register the `updateUI()` function within our store, inside `loadUI()`:

Registering updateUI with the store

```
store.subscribe(updateUI);
```

Now it's time to connect this UI logic into our application. We can do that by importing the `loadUI()` function from the file we've just created and calling it inside the *app.js* file:

app.js

```
import store from './store';
import loadUI from './ui';
import { addRecipe } from './actions/recipes';
import { addIngredient } from './actions/ingredients';

loadUI();

store.dispatch(addRecipe('Pancakes'));
store.dispatch(addIngredient('Pancakes', 'Egg', 3));
```

If you've done everything correctly, you should now see the UI in your browser.

To support adding recipes, we will add a simple input field and button and use our store's `dispatch()` method together with the `addRecipe()` action creator to send actions to the store:

Adding support for click events

```
import $ from 'jquery';
import store from './store';
import { addRecipe } from './actions/recipes';

function updateUI() {
  const { recipes } = store.getState();
  const renderRecipe = (recipe) => `<li>${ recipe.name }</li>`;

  $('.recipes > ul').html(recipes.map(renderRecipe));
}

function handleAdd() {
  const $recipeName = $('.recipes > input');

  store.dispatch(addRecipe($recipeName.val()));

  $recipeName.val('');
}

export default function loadUI() {
  $('#app').append(`
    <div class="recipes">
      <h2>Recipes:</h2>
      <ul></ul>
      <input type="text" />
      <button>Add</button>
    </div>
  `);

  store.subscribe(updateUI);

  $(document).on('click', '.recipes > button', handleAdd);

  updateUI();
}
```

Logging

Now that our UI allows us to add new recipes, we find that it's hard to see what actions are sent to the store. One option is to log received actions from the root reducer—but as we will see shortly, this can be problematic. Another option is to use the *middleware* we discussed in the previous chapter.

The store holds connections to all the middleware functions, which receive actions before the reducers. This means they have access to any actions dispatched to the store. To test this, let's create a simple logging middleware that will print any action sent to the store.

Go ahead and create a *middleware* directory and a *log.js* file within it:

middleware/log.js

```
const logMiddleware = ({ getState, dispatch }) => (next) => (action) => {
  console.log(`Action: ${ action.type }`);

  next(action);
};

export default logMiddleware;
```

The structure might seem strange at first, as we are creating a function that returns a function that returns a function. While this might be a little confusing, it is required by the way Redux combines middleware. In practice, in the innermost function we have access to the `dispatch()` and `getState()` methods from the store, the current action being processed, and the `next()` method, which allows us to call the next middleware in line.

Our logger prints the current action and then calls `next(action)` to pass the action on to the next middleware. In some cases, middleware might suppress actions or change them. That is why implementing a logger in a reducer is not a viable solution: some of the actions might not reach it.

To connect the middleware to our store, we need to modify our *store.js* file to use Redux's `applyMiddleware()` utility function:

Connecting middleware to the store

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers/root';
import logMiddleware from './middleware/log';

const store = createStore(
  rootReducer,
  applyMiddleware(logMiddleware)
);

window.store = store;

export default store;
```

Getting Data from the Server

Fetching data from a server, like anything with Redux, happens as a result of a dispatched action. In our case, the UI should dispatch an action when it loads to ask Redux to bring data to the store.

For this we will need to add a new constant to *constants/actionTypes.js* and a new action creator in *actions/recipes.js*. Our action will be called `FETCH_RECIPES`:

fetchRecipes() action creator

```
export const fetchRecipes = () => ({
  type: FETCH_RECIPES
});
```

Sadly, we can't handle the action inside a reducer. Since the action requires server access that might take time, our reducer will not be able to handle the response—reducers should return the modified state immediately.

Luckily, we have middleware, which has access to the store and thus the store's `dispatch()` method. This means we can catch the action in middleware, submit a network request, and then send a new action to the reducers with the data already inside.

Once the data arrives we will dispatch a special `SET_RECIPES` action that will pass the response from the server to the recipes reducer, which will in turn update the state using the data provided by the action.

Why do we need two actions? The first action, `FETCH_RECIPES`, is sent by the UI to tell Redux that a server API request needs to be performed. After the request is successfully completed, the data received from the server is saved to the store with a new `SET_RECIPES` action.

Let's create a `SET_RECIPES` action type in the *constants/actionTypes.js* file and add a `setRecipes()` action creator to *actions/recipes.js*:

setRecipes() action creator

```
export const setRecipes = (recipes) => ({
  type: SET_RECIPES, recipes
});
```

Here is a simple API middleware that listens to `FETCH_RECIPES` and dispatches `SET_RECIPES` when the data arrives. We will put it in a separate *middleware/api.js* file:

middleware/api.js

```
import axios from 'axios';
import { FETCH_RECIPES } from '../constants/actionTypes';
import { setRecipes } from '../actions/recipes';

const URL = 'https://s3.amazonaws.com/500tech-shared/db.json';

function fetchData(url, callback) {
  axios.get(url)
    .then(callback)
    .catch((err) => console.log(`Error fetching recipes: ${ err }`))
}

const apiMiddleware = ({ dispatch }) => next => action => {
  if (action.type === FETCH_RECIPES) {
    fetchData(URL, ({ data }) => dispatch(setRecipes(data.recipes)));
  }

  next(action);
};

export default apiMiddleware;
```

The main code of our middleware is a simple `if` statement that calls the `fetchData()` function and passes it a callback that dispatches `setRecipes()` with the returned data:

Catching API requests

```
if (action.type === FETCH_RECIPES) {
  fetchData(URL, data => store.dispatch(setRecipes(data.recipes)));
}
```

To make this middleware work, we need to add it to our store:

Adding the API middleware to the store

```
import { createStore, applyMiddleware } from 'redux';
import rootReducer from './reducers/root';
import logMiddleware from './middleware/log';
import apiMiddleware from './middleware/api';

const store = createStore(
  rootReducer,
  applyMiddleware(logMiddleware, apiMiddleware)
);

window.store = store;
export default store;
```

We also need to modify our *reducers/recipes.js* to support the new `SET_RECIPES` action:

Adding support for SET_RECIPES in the recipes reducer

```
import { ADD_RECIPE, SET_RECIPES } from '../constants/actionTypes';

const recipesReducer = (recipes = [], action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return recipes.concat({ name: action.name });

    case SET_RECIPES:
      return action.recipes;
  }

  return recipes;
};

export default recipesReducer;
```

The code for the reducer is surprisingly simple. Since we get a new list of recipes from the server, we can just return that list as the new `recipes` list:

Simple SET_RECIPES implementation

```
case SET_RECIPES:
  return action.recipes;
```

Finally, let's replace our `dispatch()` calls in *app.js* with the new `fetchRecipes()` action creator:

app.js

```
import store from './store';
import { fetchRecipes } from './actions/recipes';
import loadUI from './ui';

loadUI()

store.dispatch(fetchRecipes());
```

If you were curious enough to check the server response for the `FETCH_RECIPES` action, you might have noticed that it includes not only `recipes`, but also `ingredients`. We encourage you to take some time and implement `setIngredients()` to practice everything you have learned in this chapter. The final application is available for your reference on GitHub in the full example repository[24].

In a real application the API middleware will be more generic and robust. We will go into much more detail in the Middleware chapter in Part 2.

Summary

In this chapter we have built a simple Redux application that supports multiple reducers, middleware, and action creators. We've set up access to a server and built a minimal UI using jQuery. Now you should have enough Redux knowledge to dive deeper into more advanced aspects of various Redux actors and APIs.

[24]https://github.com/redux-book/chapter2

Part 2. Redux in Depth

Chapter 3. The Store

In contrast to most other Flux implementations, in Redux there is a single store that holds all of the application state in one object. The store is also responsible for changing the state when something happens in our application. In fact, we could say that Redux is the store. When we talk about accessing the state, dispatching an action, or listening to state changes, it is the store that is responsible for all of it.

Sometimes the concern is raised that storing the whole state in one huge JavaScript object might be wasteful. But since objects are reference-type values and the state is just an object holding references to other objects, it doesn't have any memory implications; it is the same as storing many objects in different variables.

Creating a Store

To create the store we use the `createStore()` factory function exported by Redux. It accepts three arguments: a mandatory reducer function, an optional initial state, and an optional store enhancer. We will cover store enhancers later in this chapter and will start by creating a basic store with a dummy reducer that ignores actions and returns the state as it is:

Sample store

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

const initialState = {
  recipes: [],
  ingredients: []
};

const reducer = (state, action) => state;

const store = createStore(reducer, initialState);
```

The `initialState` parameter is optional, and usually we delegate the task of building an initial state to reducers, as described in the Reducers chapter. However, it can be useful when you want to load the initial state from the server to speed up page load times. State management is covered extensively in the State Management chapter.

The simple store that we have just created has five methods that allow us to access, change, and observe the state. Let's examine each of them.

Accessing the State

The `getState()` method returns the reference to the current state:

Getting the current state from the store

```
store.getState();
// => { recipes: [], ingredients: [] }
```

Changing the State

Redux does not allow external changes to the state. Using the state object received from `getState()` and changing values directly is prohibited. The only way to cause a change of the state is by passing actions to the reducer function. Actions sent to the store are passed to the reducer and the result is used as the new global state.

Sending action objects to reducers is done via the store using the store's `dispatch()` method, which accepts a single argument, the action object:

Dispatching an action to the store

```
const action = { type: 'ADD_RECIPE', ... }
store.dispatch(action);
```

Now we can rewrite our reducer to make it able to create an updated state for actions of type `'ADD_RECIPE'` and return the current state otherwise:

Sample reducer code

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'ADD_RECIPE':
      return Object.assign(...);
    default:
      return state;
    }
  };
```

Listening to Updates

Now that we know how the store updates the state, we need a way to update the UI or other parts of our application when the state changes. The store allows us to subscribe to state changes using the `subscribe()` method. It accepts a callback function, which will be executed after every action has been dispatched:

The subscribe() method

```
const store = createStore((state) => state);

const onStoreChange = () => console.log(store.getState());

store.subscribe(onStoreChange);
```

The callback in `subscribe()` does not receive any arguments, so in order to access the state we must call `store.getState()` ourselves.

The return value of the `subscribe()` method is a function that can be used to unsubscribe from the store. It is important to remember to call `unsubscribe()` for all subscriptions to prevent memory leaks:

Unsubscribing from store updates

```
const unsubscribe = store.subscribe(onStoreChange);

// When the subscription is no longer needed
unsubscribe();
```

Replacing the Reducer

When creating the Redux store, a reducer function (often referred to as the *root reducer*) is passed as the first parameter to `createStore()`. During runtime we can use the store's `replaceReducer()` method to replace this reducer function. Usually this method is used in development to allow hot replacement of reducers. In complex applications it might be used to dynamically decorate or replace the root reducer to allow code splitting.

Code splitting—separating the production bundle into multiple files and loading them on demand when the user performs some interaction or visits a specific route—is commonly done if the application is too large to bundle into a single file or if you want to gain extra performance. Implementing lazy loading is outside the scope of this book, but you might want to know that code splitting also can be applied to the Redux store, thanks to the `replaceReducer()` method.

Let's look at a simple example with some functionality available only for authenticated users. At initial load, our store will only handle the `currentUser` substate—just enough for the authentication:

Basic reducer setup

```
import { combineReducers } from 'redux';
import { currentUser } from 'reducers/current-user';

const reducer = combineReducers({ currentUser });

const store = createStore(reducer);
```

The `combineReducers()` function is discussed in detail in the Reducers chapter.

After the user signs in, we load the new functionality. Now we need to make sure our store is aware of the new subset of our application state and the function that should handle it. Here is where the `replaceReducer()` method comes in handy:

Replacing the root reducer

```
const newReducer = combineReducers({ currentUser, recipes });

store.replaceReducer(newReducer);
```

Keep in mind that when you call `replaceReducer()`, Redux automatically calls the same initial action it calls when you first create the store, so your new reducer automatically gets executed and the new state is immediately available via the `store.getState()` method. For more on the initial action, see the Reducers chapter.

The same technique can be used by development tools to implement the hot reloading mechanism for a better developer experience. Hot reloading is a concept where source code changes don't cause a full page reload, but rather the affected code is swapped in place by special software and the application as a whole is kept in the same state that it was in before the code change. Hot reloading tools are outside the scope of this book, but you can easily find more information online.

Store as Observable

Starting from version 3.5.0, the Redux store can also act as an `Observable`. This allows libraries like RxJS to subscribe to the store's state changes. This subscription method is different from the regular `subscribe()` method of the store: when subscribing to the store as an `Observable`, the latest state is passed without the need to call `store.getState()`.

To support older browsers, Redux uses the `symbol-observable`[25] polyfill when `Symbol.observable` is not natively supported.

Here's how it works:

Integration with RxJS

```
import store from "store";
import { Observable } from "rxjs";

const store$ = Observable.from(store);

store$.forEach((state) => console.log(state))
```

This basic API is interoperable with most common Reactive libraries (e.g., RxJS). Any library that exports the `next()` method can subscribe and receive updates. This implementation also conforms to the ECMAScript `Observable` proposal[26].

If you don't use Reactive libraries, you can still subscribe to the store by accessing the `Symbol.observable` property (or using the `symbol-observable` polyfill, like Redux does):

Getting the Redux store observable

```
const observable = store[Symbol.observable]();
```

Subscribing with a generic observer will cause the observer's `next()` method to be called on every state change and passed the current store state:

Subscribing to changes

```
const observer = {
  next(state) {
    console.log("State change", state);
  }
};

const observable = store.$$observable();

const unsubscribe = observable.subscribe(observer);
```

[25] https://github.com/blesh/symbol-observable
[26] https://github.com/tc39/proposal-observable

To unsubscribe, we simply call the function returned from the call to `subscribe()`:

Unsubscribing from changes

```
const observable = store[Symbol.observable]();
const unsubscribe = observable.subscribe(observer);

unsubscribe();
```

Full Store API

The five methods of the store are:

1. `getState()`
2. `dispatch()`
3. `subscribe()`
4. `replaceReducer()`
5. `Symbol.observable`

Together they make up the whole of the store's API and most of the API Redux exports. In the first chapter of this book we learned how the first three are enough to build a fully functional application. This small API footprint, coupled with strong versatility, is what makes Redux so compelling and easy to understand.

Decorating the Store

The basic store functionality in Redux is very simple and sufficient for most use cases. Yet sometimes it is useful to slightly change one of the methods, or even add new ones to support more advanced use cases. This can be done by decorating the store. In previous versions of Redux (prior to 3.1.0) higher-order functions were used to decorate the store, but because of complicated syntax, the `createStore()` API has changed and it now supports an optional parameter for a store decorator.

Higher-Order Functions

Before we proceed, let's do a quick overview of what higher-order functions are. We can define the term as referring to a function that either takes one or more functions as arguments, returns a function as its result, or does both. Here's an example:

Sample higher-order function

```
function output(message) {
  console.log(message);
}

function addTimeStamp(fn) {
  return function(...args) {
    console.log(`Executed at: ${Date.now()}`);
    fn(...args);
  }
}

const timedOutput = addTimeStamp(output);

timedOutput('Hello World!');

> Executed at: 1464900000001
> Hello World!
```

Here, the `output()` function prints our message to the console. The `addTimeStamp()` function is a higher-order function that can take any other function as an argument and logs the time of execution. Calling `addTimeStamp()` with `output()` as the parameter creates a new "wrapped" function that has enhanced functionality. It still has the signature of the original `output()` function but now also prints the timestamp.

The compose() Function

The use of higher-order functions is very common in Redux applications, as it allows us to easily extend the behavior of other parts of the code. Take a look at an imaginary example where we have to wrap our original function in three wrappers:

Multiple decoration

```
const wrapped = third(second(first(originalFn)));

wrapped();
```

Using multiple decorators is a valid and practical approach, but the resulting code can be hard to read and appear somewhat cumbersome. Instead, Redux provides the `compose()` function to handle multiple wrappers in a cleaner manner:

Multiple wrappers with compose()

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

const wrapped = compose(third, second, first)(originalFn);

wrapped();
```

The simplest implementation of that function is very neat. Notice the use of the `reduceRight()` method on the array of functions, which ensures that wrapping of higher-order functions happens from right to left:

compose() implementation

```
function compose(...funcs) {
  return (...args) => {
    const last = funcs[funcs.length - 1]
    const rest = funcs.slice(0, -1)

    return rest.reduceRight(
      (composed, f) => f(composed),
      last(...args)
    )
  }
}
```

Store Enhancers

Store enhancers are higher-order functions used to enhance the default behavior of the Redux store. In contrast to middleware and reducers, they have access to all internal store methods (even those not available to middleware, such as `subscribe()`).

To give a few examples, there are store enhancers for:

- Store synchronization (between browser tabs or even network clients)
- State persistence
- Integration with developer tools

Let's build an example store enhancer that will persist state changes and load initial state from `localStorage`. To enhance or decorate a store, we need to write a higher-order function that receives the original store factory function as a parameter and returns a function similar in signature to `createStore()`:

Store enhancer implementation

```
import { createStore } from 'redux';
import { rootReducer } from 'reducers/root';

const persistStore = (next) =>
  (reducer, initialState, enhancer) => {
    let store;

    if (typeof initialState !== 'function') {
      store = next(reducer, initialState, enhancer);
    } else {
      const preloadedState = initialState ||
        JSON.parse(localStorage.getItem('@@PersistedState') || {})

      store = next(reducer, preloadedState, enhancer);
    }

    store.subscribe(() => localStorage.setItem(
      '@@PersistedState',
      JSON.stringify(store.getState())
    ));

  return store;
}
```

We start by creating a store. We first check if an `initialState` was originally passed. If it was, we create the store with it. Otherwise, we read the initial state from `localStorage`:

Checking if initialState was provided

```
let store;

if (typeof initialState !== 'function') {
  store = next(reducer, initialState, enhancer);
} else {
  const preloadedState = initialState ||
    JSON.parse(localStorage.getItem('@@PersistedState') || {})

  store = next(reducer, preloadedState, enhancer);
}
```

Right after our store is initiated, we subscribe to state updates and save the state to `localStorage` on every change. This will ensure our state and `localStorage` are always in sync. Finally, we return the decorated store:

Syncing localStorage with the latest state

```
store.subscribe(() => localStorage.setItem(
  '@@PersistedState',
  JSON.stringify(store.getState())
));

return store;
```

This example doesn't handle errors or edge cases, but it shows the basics of a store enhancer. To simplify the syntax, Redux allows us to pass the store enhancer as a parameter to `createStore()`:

Passing a store enhancer as an argument

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

const store = createStore(rootReducer, persistStore);
```

If you've been following along carefully, you may have noticed that in the sample code at the beginning of the chapter the second parameter to the `createStore()` function was `initialState`. Both parameters are optional, and Redux is smart enough to distinguish between the state object and the store enhancer when you pass only two arguments. However, if you also need an initial state, the parameters of `createStore()` should come in the following order:

Passing initial state before the store enhancer

```
createStore(rootReducer, initialState, persistStore);
```

We can use multiple store enhancers to create the final store for our application, and the same `compose()` method we saw earlier can be used for store composition as well:

Using compose() to combine store enhancers

```
const storeEnhancers = compose(/* ...other decorators, */ decorateStore);
const store = createStore(rootReducer, storeEnhancers);

store.createdAt // -> timestamp
```

applyMiddleware()

One of the best-known store enhancers is `applyMiddleware()`. This is currently the only store enhancer provided by Redux (we'll look at middleware in more detail in the Middleware chapter):

applyMiddleware() implementation

```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    var store = createStore(reducer, preloadedState, enhancer)
    var dispatch = store.dispatch
    var chain = []

    var middlewareAPI = {
      getState: store.getState,
      dispatch: (action) => dispatch(action)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    dispatch = compose(...chain)(store.dispatch)

    return { ...store, dispatch }
  }
}
```

In this example, the word “middlewares” is used to distinguish the plural form from the singular form in the `map()` function. This is the actual source code of `applyMiddleware()`.

At its core, `applyMiddleware()` changes the store’s default `dispatch()` method to pass the action through the chain of middleware provided:

Middleware API setup

```
var store = createStore(reducer, preloadedState, enhancer)
var dispatch = store.dispatch
var chain = []

var middlewareAPI = {
  getState: store.getState,
  dispatch: (action) => dispatch(action)
}
```

First a store is created, and the core `getState()` and `dispatch()` methods are wrapped into something called `middlewareAPI`. This is the object our middleware receives as the first parameter (commonly confused with `store`):

Building the middleware chain

```
chain = middlewares.map(middleware => middleware(middlewareAPI))
```

The array of middleware is transformed into the result of calling `middleware()` with `middlewareAPI` as its argument. Since the structure of a middleware is `api => next => action => {}`, after the transformation `chain` holds an array of functions of type `next => action => {}`.

The last stage is to use the `compose()` function to decorate the middleware one after another:

Building the dispatch chain

```
dispatch = compose(...chain)(store.dispatch)
```

This line causes each middleware to decorate the chain of previous ones in a fashion similar to this:

Composing middleware without compose()

```
middlewareA(middlewareB(middlewareC(store.dispatch)));
```

The original `store.dispatch()` is passed as a parameter to the first wrapper in the chain.

This implementation explains the strange syntax of the Redux middleware (the three-function structure):

```
const myMiddleware =
  ({ getState, dispatch }) => (next) => (action) => { ... }
```

Other Uses

Store enhancers are powerful tools that allow us to debug stores, rehydrate state on application load, persist state to `localStorage` on every action, sync stores across multiple tabs or even network connections, add debugging features, and more. If you'd like an idea of what's available, Mark Erikson has composed a list of third-party store enhancers[27] in his awesome redux-ecosystem-links[28] repository.

Summary

In this chapter we learned about the central, most important part of Redux: the store. It holds the whole state of the application, receives actions, passes them to the reducer function to replace the state, and notifies us on every change. Basically, you could say that Redux *is* the store implementation.

We learned about higher-order functions, a very powerful functional programming concept that gives us incredible power to enhance code without touching the source. We also covered uses for store enhancers in Redux, and took a look at the most common store enhancer, `applyMiddleware()`. This function allows us to intercept and transform dispatched actions before they are propagated to reducers, and we will take a deeper look at it in the Middleware chapter.

In the next chapter we will look at actions and action creators, the entities we dispatch to the store to make changes to the application state.

[27] https://github.com/markerikson/redux-ecosystem-links/blob/master/store.md
[28] https://github.com/markerikson/redux-ecosystem-links

Chapter 4. Actions and Action Creators

In Redux, changes to the global state are not done directly but rather encapsulated in "actions." This approach allows us to more easily understand the cause of changes and control the flow.

Actions can be triggered by events like keypresses or mouse clicks, timers, or network events. The receiver of an action could be a middleware or a reducer.

A connection between a sender and a receiver is not necessarily one-to-one. A keypress might cause a single action to be sent that will in turn cause both a middleware to send a message to the server and a reducer to change the state, resulting in a pop-up appearing. This also holds true in the other direction, where a single reducer might be listening to multiple actions. While in very simple Redux applications there might be a reducer for each action and vice versa, in large applications this relation breaks down, and we have multiple actions handled by a single reducer and multiple reducers and middleware listening to a single action.

Consider a logout action which is usually caught by multiple reducers, each cleaning its own part of the state.

Since the side emitting the actions doesn't know who might be listening to it, our actions have to carry all the information needed for the receiving end to be able to understand how to respond.

The simplest way to hold information in JavaScript is to use a plain object, and that is exactly what a Redux action is:

Plain object-based action

```
const action = { type: 'MARK_FAVORITE' };
```

Actions are plain objects with one required property, `type`. The `type` is a unique key describing the action, and it is used by the receiving end to distinguish between actions.

The value of the `type` property can be anything, though it is considered good practice to use strings to identify actions. While on first thought numbers or ES2015 symbols might sound like a better solution, both have practical downsides: using numbers makes it hard to debug an application and gives little benefit spacewise, whereas ES2015 symbols will cause issues with server rendering and sending actions across the network to other clients.

In Redux, we send actions to the store, which passes them to middleware and then to reducers. In order to notify a store about an action, we use the store's `dispatch()` method.

Unlike many Flux implementations, in Redux the store's `dispatch()` API is not globally available. You have a few options to access it:

1. By holding a reference to the store
2. Inside middleware
3. Through methods provided by special libraries for different frameworks

Here's a simple example of dispatching actions by holding a direct reference to the store:

Dispatching a simple action

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

const reducer = (state, action) => state;

const store = createStore(reducer);

store.dispatch({ type: 'MARK_FAVORITE' });
```

Passing Parameters to Actions

While the `type` property in an action is enough for reducers and middleware to know what action to process, in most cases more information will be useful or required. For example, instead of sending the `INCREMENT_COUNTER` action 10 times, we could send a single action and specify `10` as an additional parameter. Since actions in Redux are nothing more than objects, we are free to add as many properties as needed. The only limitation is that the `type` property is required by Redux. Suppose we decide to add a `recipeId` property as shown here:

Standard action object

```
store.dispatch({
  type: 'MARK_FAVORITE',
  recipeId: 21
  ...
});
```

The whole object passed to `dispatch()` will be available to our reducers:

Accessing actions in reducers

```
const reducer = (state, action) => {
  console.log(action);
  return state;
}

// -> { type: 'MARK_FAVORITE', recipeId: 21, ... }
```

To keep our actions consistent across a large code base, it is a good idea to define a clear scheme for how the action objects should be structured. We will discuss the scheme later in this chapter.

Action Creators

As our applications grow and develop, we will start encountering more and more code like this:

Direct object dispatch

```
dispatch({
  type: 'ADD_RECIPE',
  title: title.trim(),
  description: description ? description.trim() : ''
});
```

If we decide to use the same action in other places, we will end up having to duplicate the logic in multiple locations and remember all the parameters. This code will be hard to maintain, as we will have to synchronize the changes between all the occurrences of the action.

A better approach is to keep the code in one place. We can create a function that will create the action object for us:

Function to create an action object

```
const addRecipe = (title, description = '') => ({
  type: 'ADD_RECIPE',
  title: title.trim(),
  description:description.trim()
});
```

Now we can use it in our code:

Using an action creator

```
dispatch(addRecipe(title, description));
```

Any modification to the content or logic of the action can now be handled in one place: the action creation function, also known as the *action creator*.

Beyond improving the maintainability of our applications, moving the action object creation into a function allows us to write simpler tests. We can test the logic separately from the places where the function is called.

In a large project, most—if not all—of our actions will have a corresponding action creator function. We will try to never call the `dispatch()` method by manually creating the appropriate action object, but rather use an action creator. This might appear to be a lot of overhead initially, but its value will become apparent as the project grows. Also, to help reduce boilerplate, there are a number of libraries and concepts that ease the creation and use of action creators. Those will be discussed later in this chapter.

Directory Organization

Since the same action might be sent from multiple parts of our code, it becomes apparent that we need a place to share the action creator functions. A common pattern is to create a separate directory for all the action creators in the code base. For smaller projects, it may be enough to group actions in files according to their usage in reducers:

Simple directory structure

```
actions/
  recipes.js  // Recipe manipulation actions
  auth.js    // User actions (login, logout, etc.)
  ...
```

But as our projects grow in size and complexity, we will subdivide our action creator directory structure even more. A common approach is to nest actions based on the data type they modify:

Advanced directory structure

```
actions/
  recipes/
    favorites.js  // Handle favorite recipe logic
    ...
  auth/
    resetting-password.js  // Handle password logic
    permissions.js  // Some actions for permissions
    ...
  ...
```

At first glance it might look easier to put action creators with their corresponding reducers, sometimes even in the same files. But while this approach might work perfectly at the beginning, it will soon start breaking down in large projects. As the complexity grows, the application might have multiple reducers acting on the same action or multiple actions watched by a single reducer. In these cases the grouping stops working, and the developer is forced to start decoupling some of the actions or moving them, ending up with the structure suggested here.

Flux Standard Actions

As projects and teams grow in size, it is important to create a convention on the structure of action objects. To this end, the open-source community has come together to create the Flux Standard Action[29] (FSA) specification. The goal is to have consistency across projects and third-party libraries.

The FSA spec defines the structure of actions and a number of optional and required properties. At its base, an action should have up to four fields:

FSA object sample

```
const action = {
  type,
  error,
  payload,
  meta
};
```

Each field has a distinct role:

- `type` is the regular Redux action identifier.
- `error` is a Boolean that indicates whether the action is in an error state. The rationale behind this is that instead of having multiple actions, like `'ADD_RECIPE_SUCCESS'` and `'ADD_RECIPE_ERROR'`, we can have only one action, `'ADD_RECIPE'`, and use the `error` flag to determine the status.
- `payload` is an object holding all the information needed by the reducers. In our example, the `title` and `description` would both be passed as the `payload` property of the action.
- `meta` is a property that holds additional metadata about the action that is not necessarily needed by the reducers, but could be consumed by middleware. We will go into detail on how the `meta` property can be used in the Middleware chapter.

[29] https://github.com/acdlite/flux-standard-action

Implemented as an FSA, our action looks like this:

FSA action example

```
store.dispatch({
  type: 'ADD_RECIPE',
  payload: {
    title: 'Omelette',
    description: 'Fast and simple'
  }
});
```

If the action were in the error state (for example, in the event of a rejected promise or API failure), the payload would hold the error itself, be it an `Error` object or any other value defining the error:

FSA in error state

```
const action = {
  type: 'ADD_RECIPE',
  error: true,
  payload: new Error('Could not add recipe because...')
};
```

String Constants

In the Example Redux Application chapter, we briefly discussed the idea of using string constants. To better illustrate the reasoning behind this approach, let's consider the problems that using strings for `type` can cause in a large code base:

1. *Spelling mistakes.* If we spell the same string incorrectly in the action or the reducer, our action will fire but result in no changes to the state. Worst of all, this will be a silent failure without any message to indicate why our action failed to produce its desired effect.
2. *Duplicates.* Another developer, in a different part of the code base, might use the same string for an action. This will result in issues that are very hard to debug, as our action will suddenly cause that developer's reducers to fire as well, creating unexpected changes to the state.

To avoid these issues, we need to ensure a unique naming convention for our actions to allow action creators and reducers to use the exact same keys for the `type` parameter. Since JavaScript doesn't have native `enum` structures, we use shared constants to achieve this goal. All the keys used for `type` are stored in a *single constants file* and imported by action creators and reducers. Using a single file allows us to rely on JavaScript itself to catch duplication errors. The file will have this form:

constants/actionTypes.js

```
export const MARK_FAVORITE = 'MARK_FAVORITE';
export const ADD_RECIPE    = 'ADD_RECIPE';
```

In large applications the naming convention will be more complicated to allow developers more freedom to create constants for different parts of the application. Even in our simple example, being able to mark both recipes and comments as favorites will require two different `MARK_FAVORITE` actions (e.g., `RECIPE_MARK_FAVORITE` and `COMMENT_MARK_FAVORITE`):

constants/actionTypes.js

```
// Recipes
export const ADD_RECIPE            = 'ADD_RECIPE';
export const DELETE_RECIPE         = 'DELETE_RECIPE';
export const RECIPE__MARK_FAVORITE = 'RECIPE__MARK_FAVORITE';

// Comments
export const ADD_COMMENT            = 'ADD_COMMENT';
export const DELETE_COMMENT         = 'DELETE_COMMENT';
export const COMMENT__MARK_FAVORITE = 'COMMENT__MARK_FAVORITE';
```

Full Action Creators Example

Here's the full example code for using action creators:

constants/actionTypes.js

```
export const MARK_FAVORITE = 'MARK_FAVORITE';
```

actions/recipes.js

```
import { MARK_FAVORITE } from 'constants/actionTypes';

const markFavorite = (recipeId) => ({
  type: MARK_FAVORITE,
  recipeId,
  timestamp: Date.now()
});

export markFavorite;
```

components/recipe.js

```
import { markFavorite } from 'actions/recipes';
import store from 'store';

const handleClick = (event) => {
  store.dispatch(markFavorite(event.data.id));
};

$("button").on("click", { id: 100 }, handleClick);
```

Testing Action Creators

Since action creators are nothing more than plain JavaScript functions that should require nothing but constants and do not rely on state, they are very easy to test. Given the same input, they will always return the same output (unlike, for example, functions that read the `localStorage` or other state data that might change). Action creators also never modify any state directly, but rather only return a new object. Thus, our tests can pass input values to action creators and verify that the correct object is returned for each case:

Action creator test example

```
import { ADD_RECIPE } from 'constants/actionTypes';
import { addRecipe } from 'actions/recipes';

describe('actions/recipes', () => {
  it ('addRecipe', () => {
    const title = 'Omelette';
    const description = 'Fast and simple';
    const expectedAction = {
      type: ADD_RECIPE,
      title,
      description
    };

    expect(addRecipe(title, description)).to.equal(expectedAction);
  });

  it ('addRecipe with default title', () => {
    const description = 'Fast and simple';
    const expectedAction = {
      type: ADD_RECIPE,
      title: 'Untitled',
      description
    };

    expect(addRecipe(null, description)).to.equal(expectedAction);
  });
});
```

More complex action creators might use helper functions to build the action object. A simple example might be a helper function `trimTitle()` that removes whitespace from around a string and is used by a `SET_TITLE` action. We would test the method separately from the action creator function itself, only verifying that the action creator called that method and passed it the needed parameters.

redux-thunk

The real power of actions comes with the use of various middleware (discussed further in the Middleware chapter). One of the most common and useful ones for learning Redux is `redux-thunk`[30]. In contrast to what we've said before, actions passed to `dispatch()` don't *have* to be objects—the only part of Redux that requires actions to be objects is the reducers. Using middleware, we can transform non-object actions into objects, before they reach the reducers.

With the use of `redux-thunk` we can create asynchronous action creators that have access to both the Redux state and the `dispatch()` function.

Adding redux-thunk

Adding `redux-thunk` to a project takes two steps:

1. Add the middleware to your project by running `npm install --save redux-thunk`.
2. Load the middleware by adding it to the store using the `applyMiddleware()` function:

store/store.js

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from 'reducers/index';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);

export default store;
```

[30] https://github.com/gaearon/redux-thunk

With `redux-thunk` installed, we can start writing action creators that return functions instead of objects. Going back to our previous example, let's create an action creator that simulates server communication by using a delay:

Sample thunk action

```
import { trimmedTitle } from 'utils/strings';
import { ADD_RECIPE_STARTED } from 'actions/recipes';

function addRecipe(title)  {
  return function (dispatch, getState) {
    const trimmedTitle = trimTitle(title);

    dispatch({ type: ADD_RECIPE_STARTED });

    setTimeout(
      () => dispatch({ type: ADD_RECIPE, title: trimmedTitle }),
      1000
    );
  }
}
```

The main difference from our previous action creators is that we are now returning a function:

Sample thunk action

```
function addRecipe(title) {
  return function(dispatch, getState) {
    // Action creator's code
  };
}
```

Or, if we use the one-liner ES2015 arrow function style:

Sample thunk action using ES2015 syntax

```
const addRecipe = (title) => (dispatch, getState) => {
  // Action creator's code
};
```

Let's examine the new action creator in detail. First, we remove all whitespace around the title:

Removing unneeded whitespace

```
const trimmedTitle = trimTitle(title);
```

Next, we dispatch an action that might show a spinner in our application by setting a *fetching* flag somewhere in our state:

Dispatch on load

```
dispatch({ type: ADD_RECIPE_STARTED });
```

The last step is to send the `ADD_RECIPE` action, but delayed by one second:

Setting a timeout to add a recipe after a delay

```
setTimeout(
  () => dispatch({ type: ADD_RECIPE, title: trimmedTitle }),
  1000
);
```

Notice that in this example, a single action creator caused the dispatch of two actions: `ADD_RECIPE_-STARTED` and `ADD_RECIPE`. This gives us a new tool, allowing us to generate as many granular actions as needed and even to dispatch no actions at all in certain conditions.

How Does redux-thunk Work?

The library is in essence a small middleware that checks the `type` of every action dispatched. When it notices the action is actually a `function`, `redux-thunk` will call it, passing it the `dispatch()` and `getState()` functions as parameters and sending the return value to the reducers.

Server Communication

We can use the `redux-thunk` method to allow simple server communication by replacing the `setTimeout()` call from the previous example with a network call to the server:

An action that goes to the server

```
import * as constants from 'constants/actionTypes';
import axios from 'axios';

const getRecipes = (limit = 100) => (dispatch) => {
  dispatch({ type: constants.FETCH_RECIPES_START, limit });

  axios.get(`recipes?limit=${ limit }`)
    .then(({ data }) => dispatch({type: constants.FETCH_RECIPES_DONE, data}))
    .catch(error => dispatch({type: constants.FETCH_RECIPES_ERROR, error}));
};

export getRecipes;
```

This code might be good for a very simple project, but it includes a lot of boilerplate that will be needed for each API call we make. We have a whole Server Communication chapter to discuss a more generic and cleaner approach to server communication.

Using State

Another feature we gain by using `redux-thunk` is access to the state when processing the action. This allows us to dispatch or suppress actions according to the current application state. For example, we can prevent actions from trying to add recipes with duplicate titles:

An action that accesses state

```
const addRecipe = (title) => (dispatch, getState) => {
  const trimmedTitle = trimTitle(title);

  // We don't allow duplicate titles
  if (getState().recipes.find(place => place.title == trimmedTitle)) {
    return; // No action is performed
  }

  dispatch({
    type: ADD_RECIPE,
    payload: { title: trimmedTitle }
  });
};
```

Two new concepts can be seen in this code. We used `getState()` to get access to the full application state and used a `return` statement that made our action creator emit no action at all:

An action that accesses state

```
if (getState().recipes.find(place => place.title == trimmedTitle)) {
  return; // No action is performed
}
```

It is important to consider where such checks are performed. If multiple actions might dispatch recipe-related manipulations, we might think it is best to do the check on the reducer level (as it is the one modifying the state). Unfortunately, there is no way for the reducer to communicate any problems back to us, and while it can prevent an action from adding a duplicate title to the list, it can't dispatch an action out. The only thing a reducer can do in this case is add the error directly to the state tree.

While this approach might work, it adds complications to the reducer and causes it to be aware of multiple parts of the state tree—in our example, not just recipes but also the notifications area—which will make it harder to test and break down to multiple reducers. Thus, it is better to have the validation logic in actions or middleware.

redux-actions

When you start writing a large number of actions, you will notice that most of the code looks the same and feels like a lot of boilerplate. There are multiple third-party libraries to make the process easier and cleaner. The `redux-actions`[31] library is one of the recommended ones, as it is simple and FSA-compliant. The library allows us to easily create new actions using the provided `createAction()` function. For example:

actions/recipes.js

```
import { createAction } from 'redux-actions';
import { ADD_RECIPE } from 'constants/actionTypes';

const addRecipePayload = (title) => ({ title });

export const addRecipe = createAction(ADD_RECIPE, addRecipePayload);
```

This code generates an action creator that will return an FSA-compliant action. The generated action creator will have functionality similar to this function:

Generated action creator

```
function addRecipe(title) {
  return {
    type: ADD_RECIPE,
    payload: {
      title
    }
  };
}
```

If the title is our only payload, we could simplify the call by omitting the second argument:

Simpler action creator

```
export const addRecipe = createAction(ADD_RECIPE);
```

[31] https://github.com/acdlite/redux-actions

The resulting action creator would be as follows:

Simpler action creator's result

```
function addRecipe(title) {
  return {
    type: ADD_RECIPE,
    payload: title
  };
}
```

We can also pass metadata to the FSA action. For that, we could include a metadata object as a third argument:

Passing metadata

```
export const addRecipe = createAction(
  ADD_RECIPE,
  (title) => ({ title }),
  { silent: true }
);
```

Or, instead of a metadata object, we could use a function to calculate the metadata based on the parameters we pass in the payload:

Dynamic metadata

```
const addRecipeMetadata = (title) => ({
  silent: true,
  notifyAdmin: title === 'Omelette'
});

export const addRecipe = createAction(
  ADD_RECIPE,
  (title) => ({ title }),
  addRecipeMetadata
);
```

The usage of the action creator is the same as before. We simply call it with the desired parameters:

Resulting object with dynamic metadata

```
dispatch(addRecipe('Belgian Waffles'));

// The dispatched object:
{
  type: 'ADD_RECIPE',
  error: false,
  payload: {
    title: 'Belgian Waffles'
  },
  meta: {
    silent: true,
    notifyAdmin: false
  }
}
```

Errors

The action creator will automatically handle errors for us if passed an `Error` object. It will generate an object with `error = true` and the payload set to the `Error` object:

Dispatching errors

```
const error = new Error('Server responded with 500');

dispatch(addRecipe(error));

// The dispatched object:
{
  type: 'ADD_RECIPE',
  error: true,
  payload: Error(...),
  meta: { ... }
}
```

createAction() Example

Here's a full example of using `createAction()`:

Full example

```
const addRecipe = createAction(
  ADD_RECIPE,
  (title, description ) => (...args),
  { silent: true }
);

const addRecipeAsync = (title, description = '') => {
  const details = { ...args };

  return (dispatch) => {
    axios.post('/recipes', {
      data: JSON.stringify(details)
    })
    .then(response => dispatch(addRecipe(response.details))),
    .catch(error => dispatch(addRecipe(new Error(error))));
  }
};
```

In this example, we use the `axios` library to determine whether to create a successful action or an error one:

Passing the promise directly to an action creator

```
.then(response => dispatch(addRecipe(response.details))),
.catch(error => dispatch(addRecipe(new Error(error))));
```

Using redux-actions with redux-promise

`redux-actions` can be used in conjunction with `redux-promise`[32] to simplify the code even more.

The `redux-promise` library can automatically dispatch FSA-compliant actions and set the `error` and `payload` fields for us. It adds the ability to dispatch a promise (not just an object or function) and knows how to automatically handle the *resolve* and *reject* functionality of promises:

Automatic handling of promises

```
const addRecipe = createAction(ADD_RECIPE);

export function addRecipeAsync(details) {
  return () => addRecipe(
    axios.post('/recipes', {
      data: JSON.stringify(details)
    })
  );
}
```

The magic part here is that we pass a promise to `addRecipe()` that will take care of creating the appropriate FSA-compliant action depending on whether the promise is resolved or rejected.

This line doesn't return data, but only modifies the promise that we return from the action creator and that the caller will send to `dispatch()`.

Summary

In this chapter we covered action creators, the fuel running our application's engine. We saw that Redux is all about simplicity—actions are plain objects and action creators are just functions returning plain objects.

In the next chapter we will look at reducers, the components of our Redux application that respond to actions.

[32] https://github.com/acdlite/redux-promise

Chapter 5. Reducers

The word *reducer* is commonly associated in computer science with a function that takes an array or object and converts it to a simpler structure—for example, summing all the items in an array. In Redux, the role of the reducer is somewhat different: reducers create a new state out of the old one, based on an action.

In essence, a reducer is a simple JavaScript function that receives two parameters (two objects—the previous state and an action) and returns an object (a modified copy of the first argument):

A simple reducer example

```
const sum = (result, next) => result + next;

[1,2,3].reduce(sum); // -> 6
```

Reducers in Redux are *pure functions*, meaning they don't have any side effects such as changing `localStorage`, contacting the server, or saving any data in variables. A typical reducer looks like this:

A basic reducer

```
function reducer(state, action) {
  switch (action.type) {

    case 'ADD_RECIPE':
    // Create new state with a recipe

    default:
      return state;
  }
}
```

Reducers in Practice

In Redux, reducers are the final stage in the unidirectional data flow. After an action is dispatched to the store and has passed through all the middleware, reducers receive it together with the current state of the application. Then they create a new state that has been modified according to the action and return it to the store.

The way we connect the store and the reducers is via the `createStore()` method, which can receive three parameters: a reducer, an optional initial state, and an optional store enhancer (covered in detail in the Store chapter).

As an example, we will use an application similar to the one used in the Example Redux Application chapter—a simple recipe book application.

Our state contains three substates:

- `recipes`—A list of recipes
- `ingredients`—A list of ingredients and quantities used in each recipe
- `ui`—An object containing the state of various UI elements

And we will support the following actions:

- `ADD_RECIPE`
- `FETCH_RECIPES`
- `SET_RECIPES`

A Simple Reducer

The simplest approach to building a reducer would be to use a large `switch` statement that knows how to handle all the actions our application supports:

Using a switch statement to build a reducer

```
switch (action.type) {
  case 'ADD_RECIPE':
    // TODO: handle add recipe action

  case 'FETCH_RECIPES':
    // TODO: handle fetch recipes action

  ...
}
```

But it is quite clear that this approach will break down fast as our application (and the number of actions) grows.

Reducer Separation

The obvious solution would be to find a way to split the reducer code into multiple files, or multiple reducers. Since `createStore()` receives only one reducer, it will be that reducer's job to call other reducers to help it calculate the new state.

The simplest method of determining how to split the reducer code is by examining the state we need to handle:

Our example state

```
const state = {
  recipes: [],
  ingredients: [],
  ui: {}
}
```

We can now create three different reducers, each responsible for part of the state:

Three different reducers

```
const recipesReducer = (state, action) => {};
const ingredientsReducer = (state, action) => {};
const uiReducer = (state, action) => {};

const reducer = (state, action) => {
  // TODO: combine the result of all substate reducers
}
```

Since each of the reducers calculates a new state (or returns the original if it does not recognize the action), we can build a new state by calling all the reducers one after another:

Reducers running in sequence

```
const recipesReducer = (state, action) => {};
const ingredientsReducer = (state, action) => {};
const uiReducer = (state, action) => {};

const reducer = (state, action) => {
  let newState = state;

  newState = recipesReducer(newState, action);
  newState = ingredientsReducer(newState, action);
  newState = uiReducer(newState, action);

  return newState;
}
```

While this approach works correctly, you might have noticed a potential problem. Why does the `recipesReducer` reducer need to access and calculate the whole state, instead of only the `recipes` substate? We can further improve our reducers by having each one act on only the substate it cares about:

Substate handling reducers

```
const recipesReducer = (recipes, action) => {};
const ingredientsReducer = (ingredients, action) => {};
const uiReducer = (ui, action) => {};

const reducer = (state, action) => {
  const newState = Object.assign({}, state, {
    recipes: recipesReducer(state.recipes, action),
    ingredients: ingredientsReducer(state.ingredients, action),
    ui: uiReducer(state.ui, action)
  });

  return newState;
}
```

With this new code, each reducer receives only the part of the state that is relevant to it and can't affect other parts. This separation proves very powerful in large-scale projects, as it means developers can rely on reducers being able to modify only the parts of the state they are connected to and never causing clashes.

Another side effect of this separation of concerns is that our reducers become much simpler. Since they no longer have to calculate the whole state, a large part of the code is no longer needed:

Recipes reducer before substate

```
const recipesReducer = (state, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return Object.assign({}, state, {
        recipes: state.recipes.concat(action.payload);
      })
    ...
  }
}
```

Recipes reducer after substate

```
const recipesReducer = (recipes, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return recipes.concat(action.payload);
    ...
  }
}
```

Combining Reducers

The technique of reducer combination is so convenient and broadly used that Redux provides a very useful function named `combineReducers()` to facilitate it. This helper function does exactly what `rootReducer()` did in our earlier example, with some additions and validations:

Root reducer using combineReducers()

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';

const rootReducer = combineReducers({
  recipes: recipesReducer,
  ingredients: ingredientsReducer
});

export const store = createStore(rootReducer);
```

We can make this code even simpler by using ES2015's property shorthand feature:

Using ES2015 syntax in combineReducers()

```
import { combineReducers } from 'redux';
import recipes from 'reducers/recipes';
import ingredients from 'reducers/ingredients';
import ui from 'reducers/ui';

export default combineReducers({
  recipes,
  ingredients,
  ui
});
```

In this example we provided `combineReducers()` with a configuration object holding keys named `recipes`, `ingredients`, and `ui`. The ES2015 syntax we used automatically assigned the value of each key to be the corresponding reducer.

It is important to note that `combineReducers()` is not limited only to the root reducer. As our state grows in size and depth, nested reducers will be combining other reducers for substate calculations. Using nested `combineReducers()` calls and other combination methods is a common practice in larger projects.

Default Values

One of the requirements of `combineReducers()` is for each reducer to define a default value for its substate. Using this approach, the default structure of the state tree is dynamically built by the reducers themselves. This guarantees that changes to the tree require changes only to the applicable reducers and do not affect the rest of the tree.

This is possible because when the store is created, Redux dispatches a special action called `@@redux/INIT`. Each reducer receives that action together with the undefined initial state, which gets replaced with the default parameter defined inside the reducer. Since our `switch` statements do not process this special action type and simply return the state (previously assigned by the default parameter), the initial state of the store is automatically populated by the reducers.

To support this, each of the subreducers must define a default value for its first argument, to use if none is provided:

Defining a default substate

```
const initialState = [];

const recipesReducer = (recipes = initialState, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return recipes.concat(action.payload);
    ...
  }
}
```

Tree Mirroring

This brings us to an important conclusion: that we want to structure our reducers tree to mimic the application state tree. As a rule of thumb, we will want to have a reducer for each leaf of the tree. Mimicking this structure in the reducers directory will make it self-depicting of how the state tree is structured.

As complicated manipulations might be required to add some parts of the tree, some reducers might not neatly fall into this pattern. We might find ourselves with two or more reducers processing the same subtree (sequentially), or a single reducer operating on multiple branches (if it needs to update structures in different branches). This might cause complications in the structure and composition of our application. Such issues can usually be avoided by normalizing the tree, splitting a single action into multiple ones, and other techniques.

Alternative to switch Statements

In Redux, most reducers are just `switch` statements over `action.type`. Since the `switch` syntax can be hard to read and prone to errors, there are a few libraries that try to make writing reducers easier and cleaner.

While it is most common for a reducer to examine the `type` property of the action to determine if it should act, in some cases other parts of the action object are used. For example, you might want to show an error notification on every action that has an error in the payload.

The `redux-actions` library described in the previous chapter provides the `handleActions()` utility function for reducer generation:

Using redux-actions for reducer creation

```
import { handleActions } from 'redux-actions';

const initialState = [];

const recipesReducer = handleActions({
  [ADD_RECIPE]: (recipes, action) =>
    recipes.concat(action.payload),

  [REMOVE_RECIPE]: (recipes, action) =>
    recipes.filter(recipe => recipe.id === action.payload)

}, initialState);

export default recipesReducer;
```

If you are using Immutable.js, you might also want to take a look at the `redux-immutablejs`[33] library, which provides you with `createReducer()` and `combineReducers()` functions that are aware of Immutable.js features like getters and setters.

Avoiding Mutations

The most important thing about reducers in Redux is that *they should never mutate the existing state*. There are a number of functions in JavaScript that can help when working with immutable objects. Before we look at those, however, let's consider why this is so important.

Why Do We Need to Avoid Mutations?

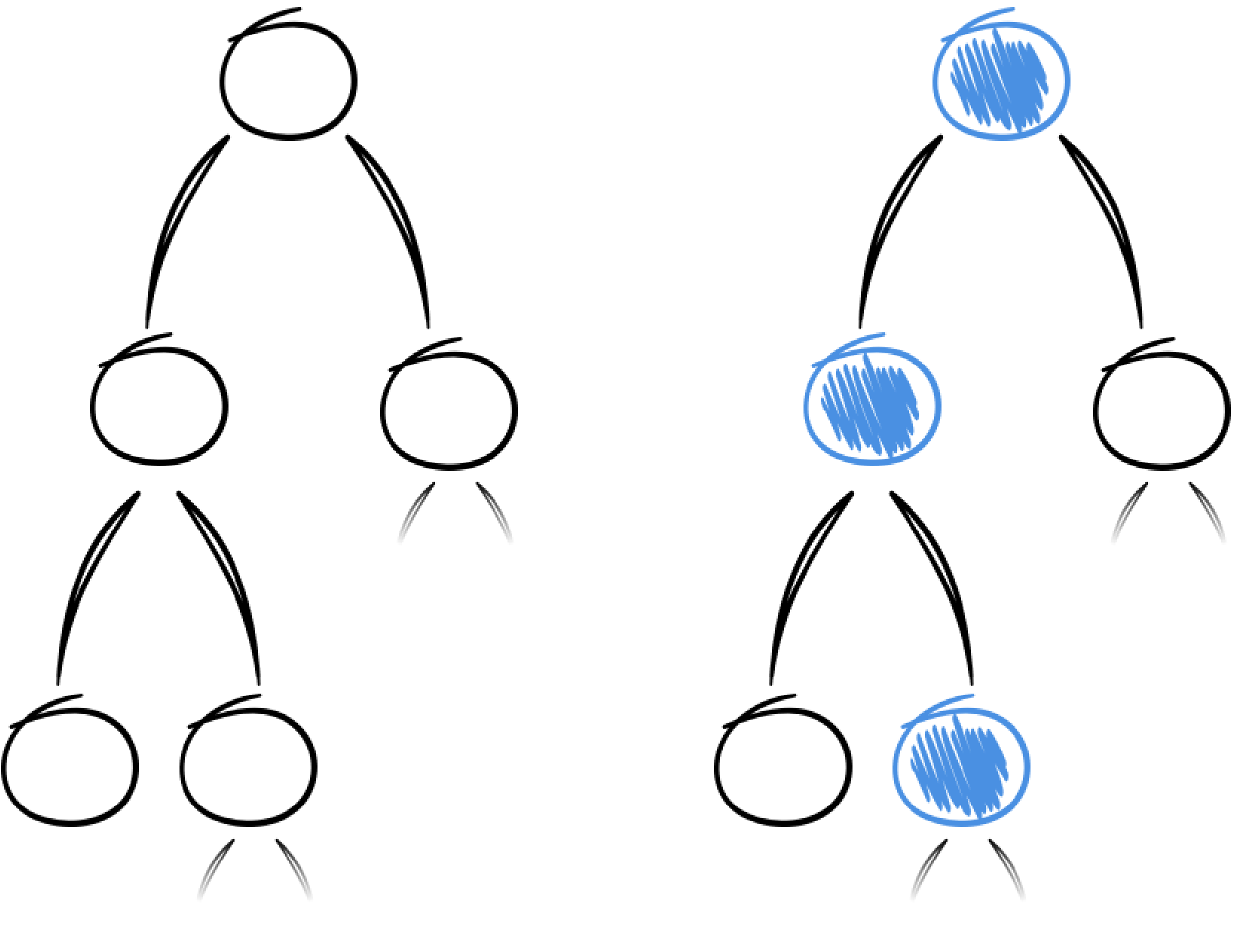
One of the reasons behind the immutability requirement for reducers is due to *change detection*. After the store passes the current state and action to the root reducer, it and various UI components of the application need a way to determine what changes, if any, have happened to the global state. For small objects, a deep compare or other similar methods might suffice. But if the state is large and only a small part may have changed due to an action, we need a faster and better method.

There are a number of ways to detect a change made to a tree, each with its pros and cons. Among the many solutions, one is to mark where changes were made in the tree. We can use simple methods like setting a `dirty` flag, use more complicated approaches like adding a version to each node, or (the preferred Redux way) use *reference comparison*.

Redux and its accompanying libraries rely on reference comparison. After the root reducer has run, we should be able to compare the state at each level of the state tree with the same level in the previous version of the tree to determine if it has changed. But instead of comparing each key and value, we can compare just the reference or *pointer* to the structure.

In Redux, each changed node or leaf is replaced by a new copy of itself that incorporates the changed data. Since the node's parent still points to the old copy of the node, we need to create a copy of it as well, with the new copy pointing to the new child. This process continues with each parent being recreated until we reach the root of the tree. This means that a change to a leaf must cause its parent, the parent's parent, etc. to be modified. In other words, it causes new objects to be created. The following illustration shows the state before and after it is run through the reducers tree and highlights the changed nodes.

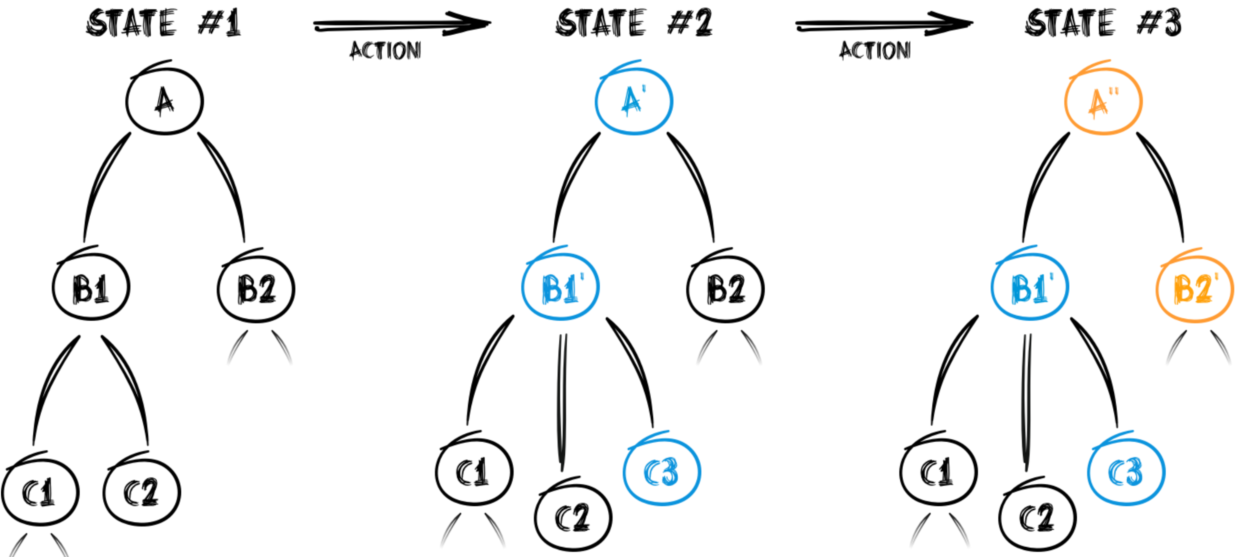
[33] https://github.com/indexiatech/redux-immutablejs



Example of changing nodes

The main reason for using reference comparison is that this method ensures that each reference to the previous state is kept coherent. We can examine the reference at any time and get the state exactly as it was before a change. If we create an array and push the current state into it before running actions, we will be able to pick any of the pointers to the previous state in the array and see the state tree exactly as it was before all the subsequent actions happened. And no matter how many more actions we process, our original pointers stay exactly as they were.

This might sound similar to copying the state each time before changing it, but the reference system will not require 10 times the memory for 10 states. It will smartly reuse all the unchanged nodes. Consider the next illustration, where two different actions have been run on the state, and how the three trees look afterward.



Example of saving references

The first action added a new node, C3, under B1. If we look closely we can see that the reducer didn't change anything in the original A tree. It only created a new A' object that holds B2 and a new B1' that holds the original C1 and C2 and the new C3'. At this point we can still use the A tree and have access to all the nodes like they were before. What's more, the new A' tree didn't copy the old one, but only created some new links that allow efficient memory reuse.

The next action modified something in the B2 subtree. Again, the only change is a new A'' root object that points to the previous B1' and the new B2'. The old states of A and A' are still intact and memory is reused between all three trees.

Since we have a coherent version of each previous state, we can implement nifty features like undo and redo (we save the previous state in an array and, in the case of "undo," make it the current one). We can also implement more advanced features like "time travel," where we can easily jump between versions of our state for debugging.

What Is Immutability?

Let's define what the word *mutation* means. In JavaScript, there are two types of variables: ones that are copied by value, and ones that are passed by reference. Primitive values such as numbers, strings, and Booleans are copied when you assign them to other variables, and a change to the target variable will not affect the source:

Primitive values example

```
let string = "Hello";
let stringCopy = string;

stringCopy += " World!";

console.log(string); // => "Hello"
console.log(stringCopy); // => "Hello World!"
```

In contrast, *collections* in JavaScript aren't copied when you assign them; they only receive a pointer to the location in memory of the object pointed to by the source variable. This means that any change to the new variable will modify the same memory location, which is pointed to by both old and new variables:

Collections example

```
const object = {};
const referencedObject = object;

referencedObject.number = 42;

console.log(object); // => { number: 42 }
console.log(referencedObject); // => { number: 42 }
```

As you can see, the original object is changed when we change the copy. We used `const` here to emphasize that a constant in JavaScript holds only a pointer to the object, not its value, and no error will be thrown if you change the properties of the object (or the contents of an array). This is also true for collections passed as arguments to functions, as what is being passed is the reference and not the value itself.

Luckily for us, ES2015 lets us avoid mutations for collections in a much cleaner way than before, thanks to the `Object.assign()` method and the *spread operator*.

The spread operator is fully supported by the ES2015 standard. More information is available on MDN[34]. Complete `Object.assign()` documentation is also available on MDN[35]. MDN is great!

Objects

`Object.assign()` can be used to copy all the key/value pairs of one or more source objects into one target object. The method receives the following parameters:

1. The target object to copy *to*
2. One or more source objects to copy *from*

Since our reducers need to create a new object and make some changes to it, we will pass a new empty object as the first parameter to `Object.assign()`. The second parameter will be the original subtree to copy and the third will contain any changes we want to make to the object. This will result in us always having a fresh object with a new reference, having all the key/value pairs from the original state and any overrides needed by the current action:

Example of Object.assign()

```
function reduce(state, action) {
  const overrides = { price: 0 };

  return Object.assign({}, state, overrides);
}

const state = { ... };
const newState = reducer(state, action);

state === newState; // false!
```

Deleting properties can be done in a similar way using ES2015 syntax. To delete the key `name` from our state we can use the following:

Example of deleting a key from an object

```
return Object.assign({}, state, { name: undefined } );
```

[34] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Spread_operator
[35] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

Arrays

Arrays are a bit trickier, since they have multiple methods for adding and removing values. In general, you just have to remember which methods create a new copy of the array and which change the original one. For your convenience, here is a table outlining the basic array methods:

Safe methods	Mutating methods
concat()	push()
slice()	splice()
map()	pop()
reduce()	shift()
reduceRight()	unshift()
filter()	fill()
	reverse()
	sort()
	copyWithin()

The basic array operations we will be doing in most reducers are appending, deleting, and modifying an array. To keep to the immutability principles, we can achieve these using the following methods:

Adding items to an array

```
const reducer = (state, action) =>
  state.concat(newValue);
```

Removing items from an array

```
const reducer = (state, action) =>
  state.filter(item => item.id !== action.payload);
```

Changing items in an array

```
const reducer = (state, action) =>
  state.map((item) => item.id !== action.payload
    ? item
    : Object.assign({}, item, { favorite: action.payload })
  );
```

Ensuring Immutability

The bitter truth is that in teams with more than one developer, we can't always rely on everyone avoiding state mutations all the time. As humans, we make mistakes, and even with the strictest pull request, code review, and testing practices, sometimes they crawl into the code base. Fortunately, there are a number of methods and tools that strive to protect developers from these hard-to-find bugs.

One approach is to use libraries like `deep-freeze`[36] that will throw errors every time someone tries to mutate a "frozen" object. While JavaScript provides an `Object.freeze()` method, it freezes only the object it is applied to, not its children. `deep-freeze` and similar libraries perform nested freezes and method overrides to better catch such errors.

Another approach is to use libraries that manage truly immutable objects. While they add additional dependencies to the project, they provide a variety of benefits as well: they ensure true immutability, offer cleaner syntax to update collections, support nested objects, and provide performance improvements on very large data sets.

The most common library is Facebook's Immutable.js[37], which offers a number of key advantages (in addition to many more advanced features):

- Fast updates on very large objects and arrays
- Lazy sequences
- Additional data types not available in plain JavaScript
- Convenient methods for deep mutation of trees
- Batched updates

It also has a few disadvantages:

- It's an additional large dependency for the project.
- It requires the use of custom getters and setters to access the data.
- It might degrade performance where large structures are not used.

It is important to carefully consider your state tree before choosing an immutable library. The performance gains might only become perceptible for a small percentage of the project, and the library will require all of the developers to understand a new access syntax and collection of methods.

[36] https://www.npmjs.com/package/deep-freeze
[37] https://facebook.github.io/immutable-js/

Another library in this space is `seamless-immutable`[38], which is smaller, works on plain JavaScript objects, and treats immutable objects the same way as regular JavaScript objects (though it has similar convenient setters to Immutable.js). Its author has written a great post[39] where he describes some of the issues he had with Immutable.js and what his reasoning was for creating a smaller library.

`seamless-immutable` does not offer many of the advantages of Immutable.js (sequences, batching, smart underlying data structures, etc.), and you can't use advanced ES2015 data structures with it, such as `Map`, `Set`, `WeakMap`, and `WeakSet`.

The last approach is to use special helper functions that can receive a regular object and an instruction on how to change it and return a new object as a result. The `immutability-helper` library[40] provides one such function, named `update()`. Its syntax may look a bit weird, but if you don't want to work with immutable objects and clog object prototypes with new functions, it might be a good option:

Example of using the update() function

```
import update from 'immutability-helper';

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

Using Immer for Temporary Mutations

> A word from our friend Michel Weststrate[41], creator of MobX[42] and Immer[43].

When writing reducers it can sometimes be beneficial to temporarily use mutable objects. This is fine as long as you only mutate new objects (and not an existing state), and as long as you don't try to mutate the objects after they have left the reducer.

[38] https://github.com/rtfeldman/seamless-immutable
[39] http://tech.noredink.com/post/107617838018/switching-from-immutablejs-to-seamless-immutable
[40] https://github.com/kolodny/immutability-helper
[41] https://twitter.com/mweststrate
[42] https://mobx.js.org
[43] https://github.com/mweststrate/immer

Immer is a tiny library that expands this idea and makes it easier to write reducers. It is comparable in functionality to the `withMutations()` method in Immutable.js, but applied to regular JavaScript structures. The advantage of this approach is that you don't have to load an additional library for data structures. Nor do you need to learn a new API to perform complex mutations in reducers. And last but not least, TypeScript and Flow are perfectly able to type-check the reducers that are created using Immer.

Let's take a quick look at Immer. The Immer package exposes a `produce()` function that takes two arguments: the *current state* and a *producer function*. The producer function is called by `produce()` with a *draft*.

The draft is a virtual state tree that reflects the entire current state. It will record any changes you make to it. The `produce()` function returns the next state by combining the current state and the changes made to the draft.

So, let's say we have the following example reducer:

Example reducer

```
const byId = (state, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return {
        ...state,
        ...action.products.reduce((obj, product) => {
          obj[product.id] = product
          return obj
        }, {})
      }

    default:
      return state
  }
};
```

This may be hard to grasp at first glance because there is quite a bit of noise, resulting from the fact that we are manually building a new state tree. With Immer, we can simplify this to:

Example reducer with Immer

```
const byId = (state, action) =>
  produce(state, draft => {
    switch (action.type) {
      case RECEIVE_PRODUCTS:
        action.products.forEach(product => {
          draft[product.id] = product
        });

        break;
    }
  });
```

The reducer will now return the next state produced by the producer. If the producer doesn't do anything, the next state will simply be the original state. Because of this, we don't have to handle the default case.

Immer will use structural sharing, just like if we had written the reducer by hand. Beyond that, because Immer knows which parts of the state were modified, it will also make sure that the modified parts of the tree will automatically be frozen in development environments. This prevents accidentally modifying the state after `produce()` has ended.

To further simplify reducers, the `produce()` function supports currying. It is possible to call `produce()` with just the producer function. This will create a new function that will execute the producer with the state as an argument. This new function also accepts an arbitrary amount of additional arguments and passes them on to the producer. This allows us to write the reducer solely in terms of the draft itself:

Simplified reducer with Immer

```
const byId = produce((draft, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      action.products.forEach(product => {
        draft[product.id] = product
      });

      break;
  }
})
```

If you want to take full advantage of Redux, but still like to write your reducers with built-in data structures and APIs, make sure to give Immer a try.

Higher-Order Reducers

The power of Redux is that it allows you to solve complex problems using functional programming. One approach is to use *higher-order functions*. Since reducers are nothing more than pure functions, we can wrap them in other functions and create very simple solutions for very complicated problems.

There are a few good examples of using higher-order reducers—for example, for implementing undo/redo functionality. There is a library called `redux-undo`[44] that takes your reducer and enhances it with undo functionality. It creates three substates: *past*, *present*, and *future*. Every time your reducer creates a new state, the previous one is pushed to the past states array and the new one becomes the present state. You can then use special actions to undo, redo, or reset the present state.

[44] https://github.com/omnidan/redux-undo

Using a higher-order reducer is as simple as passing your reducer into an imported function:

Using a higher-order reducer

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';
import undoable from 'redux-undo';

const rootReducer = combineReducers({
  recipes: undoable(recipesReducer),
  ingredients: ingredientsReducer
});
```

Another example of a reducer enhancer is `redux-ignore`[45]. This library allows your reducers to immediately return the current state without handling the passed action, or to handle only a defined subset of actions. The following example will disable removing recipes from our recipe book. You might even use it to filter allowed actions based on user roles:

Using the ignoreActions() higher-order reducer

```
import { combineReducers } from 'redux';
import recipesReducer from 'reducers/recipes';
import ingredientsReducer from 'reducers/ingredients';
import { ignoreActions } from 'redux-ignore';
import { REMOVE_RECIPE } from 'constants/actionTypes';

const rootReducer = combineReducers({
  recipes: ignoreActions(recipesReducer, [REMOVE_RECIPE]),
  ingredients: ingredientsReducer
});
```

Summary

In this chapter we learned about the part of Redux responsible for changing the application state. Reducers are meant to be pure functions that should never mutate the state or make any asynchronous calls. We also learned how to avoid and catch mutations in JavaScript.

In the next and final chapter in this part of the book we are going to talk about *middleware*, the most powerful entity provided by Redux. When used wisely, middleware can significantly reduce the size of our code and let us handle very complicated scenarios with ease.

[45]https://github.com/omnidan/redux-ignore

Chapter 6. Middleware

Middleware is one of Redux's most powerful concepts and will hold the bulk of an application's logic and generic service code. To understand middleware, it's best first to examine the regular data flow in Redux. Any action dispatched to the store is passed to the root reducer together with the current state to generate a new one. Middleware allow us to add code that will run before the action is passed to the reducer.

In essence, we can monitor all the actions being sent to the Redux store and execute arbitrary code before allowing an action to continue to the reducers. Multiple middleware can be added in a chain, with each running its own logic, one after another, before letting the action pass through.

The basic structure of a middleware is as follows:

Basic middleware structure

```
const myMiddleware = ({ getState, dispatch }) => next => action => {
  next(action);
};
```

This declaration might seem confusing, but it should become clear once it's examined step by step. At its base, a middleware is a function that receives from Redux an object that contains the `getState()` and `dispatch()` functions. The middleware returns back a function that receives `next()` and in turn returns another function that receives `action` and finally contains our custom middleware code.

From the user's perspective, a simple `const myMiddleware = ({ getState, dispatch, next, action }) => {}` might have seemed sufficient. The more complicated structure is due to Redux internals.

The `getState()` and `dispatch()` methods should be familiar as they are APIs from the Redux store. The `action` parameter is the current Redux action being passed to the store. Only the `next()` function should look unfamiliar at this point.

It is sometimes incorrectly noted in teaching material that the parameter passed to the middleware is `store`, as it has `getState()` and `dispatch()` methods. In practice, it's an object holding only those two APIs and not the other APIs exported by the Redux store.

Understanding next()

If we were to build our own implementation of middleware, we would probably want the ability to run code both before an action is passed to the reducers and after. One approach would be to define two different callbacks for before and after.

Redux middleware takes a different approach and gives us the `next()` function. Calling it with an action will cause it to propagate down the middleware chain, calling the root reducer and updating the state of the store. This allows us to add code before and after passing the action to the reducers:

Before and after next()

```
const logMiddleware => ({ getState, dispatch }) => next => action => {
  console.log("Before reducers have run");
  next(action);
  console.log("After reducers have run");
};
```

This nifty trick gives us more power than might initially be apparent. Since we are responsible for calling `next()` and passing it the action, we can choose to suppress `next()` in certain conditions or even modify the current action before passing it on. Failing to call `next(action)` inside a middleware will prevent the action from reaching the other middleware and reducers.

Our First Middleware

To demonstrate the power of middleware, let's build a special debug middleware that measures how much time it takes for our reducers to process an action.

Folder Structure

A common approach is to keep all our middleware implementations in a *middleware* directory at our application root, similar to reducers and actions. As our application grows, we might find ourselves adding more subdirectories to organize the middleware, usually by functionality (utility or authorization).

As with "software" and "hardware," we use "middleware" as both the singular and the plural form of the word. In some sources, including the code for `applyMiddleware()` shown in the Store chapter, you may see "middlewares" used as the plural form.

The measureMiddleware

Our time-measuring middleware looks like this:

middleware/measure.js

```
const measureMiddleware = () => next => action => {
  console.time(action.type);
  next(action);
  console.timeEnd(action.type);
};
```

To create this middleware we use the `time()` and `timeEnd()` console methods that record a benchmark with the name provided as a string. We start the timing before running an action, using the `action.type` as a name. Then, we tell the browser to print the timing after the action is done. This way we can potentially catch poorly performing reducer implementations.

An interesting thing to note here is that this middleware completely ignores the first parameter (the object holding `getState()` and `dispatch()`), as we don't need it for our example.

Connecting to Redux

Adding a middleware to the Redux store can be done only during the store creation process:

Regular Redux store

```
import { createStore } from 'redux';
import reducer from 'reducers/root';

const store = createStore(reducer);
```

The simplest way to connect a middleware to the Redux store is to use the `applyMiddleware()` store enhancer available as an API from Redux itself (store enhancers are explained in the Store chapter):

Creating a store and registering the measureMiddleware

```
import { createStore, applyMiddleware } from 'redux';
import reducer from 'reducers/root';
import measureMiddleware from 'middleware/measure';

const store = createStore(reducer, applyMiddleware(measureMiddleware));
```

The `applyMiddleware()` function can receive an arbitrary number of middleware as arguments and create a chain to be connected to the store:

Creating a chain of middleware

```
applyMiddleware(middlewareA, middlewareB, middlewareC);
```

Note that the order of registration is important. The first middleware, in our case `middlewareA`, will get the action before `middlewareB`. And if the code there decides to modify or suppress the action, it will never reach either `middlewareB` or `middlewareC`.

In real-world applications, you may prefer to apply some middleware only in development or production environments. For example, our `measureMiddleware` might output unwanted information in the live product, and an `analyticsMiddleware` might report false analytics in development.

Using the spread operator from ES2015, we can apply middleware to the store conditionally:

Conditionally apply middleware

```
const middleware = [apiMiddleware];

if (development) {
  middleware.push(measureMiddleware);
} else {
  middleware.push(analyticsMiddleware);
}

const store = createStore(reducer, applyMiddleware(...middleware));
```

Async Actions

What makes middleware so powerful is the access to both `getState()` and `dispatch()`, as these functions allow a middleware to run asynchronous actions and give it full access to the store. A very simple example would be an action debounce middleware. Suppose we have an autocomplete field, and we want to prevent the `AUTO_COMPLETE` action from running as the user types in a search term. We would probably want to wait 500 ms for the user to type in part of the search string, and then run the query with the latest value.

We can create a debounce middleware that will catch any action with the `debounce` key set in its metadata and ensure it is delayed by the specified number of milliseconds. Any additional action of the same type that is passed before the debounce timer expires will not be passed to reducers but only saved as the "latest action" and executed once the debounce timer has expired:

Debounce flow

```
0ms: dispatch({ type: 'AUTO_COMPLETE', payload: 'c', meta: { debounce: 500 }};
// Suppressed

10ms: dispatch({ type: 'AUTO_COMPLETE', payload: 'ca', meta: { debounce: 500 }};
// Suppressed

20ms: dispatch({ type: 'AUTO_COMPLETE', payload: 'cat', meta: { debounce: 500 }};
// Suppressed

520ms:
// The action with payload 'cat' is dispatched by the middleware.
```

The skeleton of our middleware needs to inspect only actions that have the required `debounce` key set in their metadata:

debounceMiddleware skeleton

```
const debounceMiddleware = () => next => action => {
  const { debounce } = action.meta || {};

  if (!debounce) {
    return next(action);
  }

  // TODO: Handle debouncing
};
```

Since we want each action type to have a different debounce queue, we will create a `pending` object that will hold information for each action type. In our case, we only need to save the latest timeout for each action type:

Saving the latest debounced action

```
// Object to hold debounced actions (referenced by action.type)
const pending = {};

const debounceMiddleware = () => next => action => {
  const { debounce } = action.meta || {};

  if (!debounce) {
    return next(action);
  }

  if (pending[action.type]) {
    clearTimeout(pending[action.type])
  }

  // Save latest action object
  pending[action.type] = setTimeout(/* implement debounce */);
};
```

If there is already a pending action of this type, we cancel the timeout and create a new timeout to handle this action. The previous one can be safely ignored—for example, in our case if an action `{ type: 'AUTO_COMPLETE', payload: 'cat' }` comes right after `{ type: 'AUTO_COMPLETE', payload: 'ca' }`, we can safely ignore the one with `'ca'` and only call the autocomplete API for `'cat'`:

Timeout handler

```
setTimeout(
  () => {
    delete pending[action.type];
    next(action);
  },
  debounce
);
```

Once the timeout for the latest action has elapsed, we clear the key from our `pending` object and `next()` method to allow the last delayed action to finally pass through to the other middleware and the store:

Complete debounceMiddleware

```
// Object to hold debounced actions (referenced by action.type)
const pending = {};

const debounceMiddleware = () => next => action => {
  const { debounce } = action.meta || {};

  if (!debounce) {
    return next(action);
  }

  if (pending[action.type]) {
    clearTimeout(pending[action.type]);
  }

  pending[action.type] = setTimeout(
    () => {
      delete pending[action.type];
      next(action);
    },
    debounce
  );
};
```

With this basic middleware we have created a powerful tool for our developers. A simple `meta` setting on an action can now support debouncing of any action in the system. We have also used the middleware's support for the `next()` method to selectively suppress actions. In the Server Communication chapter in Part 3 we will learn about more advanced uses of the async flow to handle generic API requests.

Using Middleware for Flow Control

One important usage of middleware is to control the application's flow and logic. Let's consider a sample user login flow:

1. Send `POST` request to server to log in
2. Save access token in store
3. Fetch current user's profile information
4. Fetch latest notifications

Usually our flow will begin with a `LOGIN` action containing the login credentials:

Example of a login action

```
{
  type: 'LOGIN',
  payload: {
    email: 'info@redux-book.com',
    password: 'will-never-tell'
  }
}
```

After our access to the server completes successfully, another action will typically be dispatched, similar to:

Successful login action

```
{
  type: 'SUCCESSFUL_LOGIN',
  payload: 'access_token'
}
```

One of the reducers will make sure to update the access token in the state, but it is unclear who is responsible for issuing the two additional actions required: `FETCH_USER_INFO` and `FETCH_NOTIFICATIONS`.

We could always use complex action creators and the `redux-thunk` middleware in our login action creator:

Complex login action creator

```
const login = (email, password) => dispatch => {
  postToServer('login', { email, password })
    .then((response) => {
      dispatch(successfulLogin(response.accessToken));
      dispatch(fetchUserInfo());
      dispatch(fetchNotifications());
  });
};
```

But this might cause a code reuse problem. If in the future we want to support Facebook Connect, it will require a different action altogether, which will still need to include the calls to `FETCH_USER_INFO` and `FETCH_NOTIFICATIONS`. And if in the future we change the login flow, it will need to be updated in multiple places.

A simple solution to the problem is to cause the two actions to be dispatched only after the login is successful. In the regular Redux flow, there is only one actor that can listen for and react to events—the middleware:

Login flow middleware

```
const loginFlowMiddleware = ({ dispatch }) => next => action => {
  // Let the reducer save the access token in the state
  next(action);

  if (action.type === SUCCESSFUL_LOGIN) {
    dispatch(fetchUserInfo());
    dispatch(fetchNotifications());
  }
};
```

Our new code holds the flow in a single place and will allow us to easily support login via Twitter, Google Apps, and more. In practice we can combine flows together and add conditions and more complicated logic, as we have full access to both `dispatch()` and `getState()`.

There are a few external libraries that try to make flow management easier, such as `redux-saga`[46].

[46] https://github.com/redux-saga/redux-saga

Other Action Types

When learning the basics of Redux, developers are usually taught that actions in Redux can only be objects and they must contain the `type` property. In practice, reading this book and online material, you'll notice `Error` objects, functions, and promises being passed to `dispatch()`.

The underlying rule is simple: our root reducer requires an object as an action, but our middleware have no such limits and are free to handle any type of data passed to `dispatch()`.

Try running `dispatch(null)`. You should get an error from Redux similar to:

Passing a non-object to dispatch()

```
store.dispatch(null);
> Uncaught TypeError: Cannot read property 'type' of null(...)
```

To add support for `null` we can create our own `nullMiddleware`:

Simple nullMiddleware

```
const nullMiddleware = () => next => action => {
  next(action !== null ? action : { type: 'UNKNOWN' });
};
```

In this middleware we catch any attempt to send `null` instead of the action object and dispatch a fake `{ type: 'UNKNOWN' }` instead. While this middleware has no practical value, it should be apparent how we can use the middleware's power to change actions to support any input type.

The famous `redux-thunk`[47] middleware is in essence the following code:

Simplified redux-thunk

```
const thunkMiddleware = ({ dispatch, getState }) => next => action => {
  if (typeof action === 'function') {
    return action(dispatch, getState);
  }

  return next(action);
};
```

It checks if the action passed is a function and not a regular object, and if it is, it calls it, passing `dispatch()` and `getState()`. A similar approach is used by other helper middleware that know how to accept promises that dispatch a regular action once a promise is resolved, use `Error` objects to report errors, use arrays to execute a list of actions provided in parallel or sequentially, and more.

[47] https://github.com/gaearon/redux-thunk

Parameter-Based Middleware

In addition to the basic middleware we've discussed so far in this chapter, some middleware might be reusable and support parameters being passed during their creation.

For example, consider the `nullMiddleware` we created in the previous section:

Simple nullMiddleware

```
const nullMiddleware = () => next => action => {
  next(action !== null ? action : { type: 'UNKNOWN' });
};

export default nullMiddleware;
```

The `'UNKNOWN'` key is hardcoded into our middleware and will not allow easy reuse in our other projects. To make this middleware more generic, we might want to be able to support arbitrary action types and use the `applyMiddleware()` stage to specify how we want our middleware to behave:

Customizable middleware

```
import { createStore, applyMiddleware } from 'redux';
import reducer from 'reducers/root';
import nullMiddleware from 'middleware/null';

const store = createStore(reduce, applyMiddleware(nullMiddleware('OH_NO'));
};
```

Here we want our `nullMiddleware` to dispatch `'OH_NO'` instead of the default `'UNKNOWN'`. To support this we must turn our middleware into a "middleware creator":

nullMiddleware creator

```
const nullMiddlewareCreator = param => () => next => action => {
   next(action !== null ? action : { type: param || 'UNKNOWN' });
};

export default nullMiddlewareCreator;
```

Now instead of returning the middleware directly, we return a function that creates a middleware with custom parameters passed in. This behavior can be further extended to allow complex middleware as libraries that can be easily customized when added to the store.

The Difference Between next() and dispatch()

A common point of confusion is the difference between the `next()` and `dispatch()` functions passed to the middleware. While both accept an action, they follow a different flow in Redux.

Calling `next()` within a middleware will pass the action along the middleware chain (from the current middleware) down to the reducer. Calling `dispatch()` will start the action flow from the beginning (the first middleware in the chain), so it will eventually reach the current one as well. Suppose we have the following middleware:

Example store with a middleware chain

```
createStore(reducer,
  applyMiddleware(middlewareA, middlewareB, middlewareC)
);
```

Calling `next(action)` within `middlewareB` results in calls to `middlewareC` and the reducer.

Calling `dispatch(action)` within `middlewareB` results in calls to `middlewareA`, `middlewareB`, `middlewareC`, and the reducer.

Calling `dispatch()` multiple times is a common and valid practice. `next()` can also be called more than once, but this is not recommended as any action passed to `next()` will skip the middleware before the current one (for example, potentially skipping the logging middleware).

How Are Middleware Used?

In real-life applications, middleware are often where most of the logic and complex code resides. They also hold most of the utility functionality (logging, error reporting, analytics, authorization, and more), and many of the enhancement libraries used in this book require the addition of their custom middleware to the chain in order to support their functionality.

There is a long list of open source projects that implement various useful middleware. A few examples are `redux-analytics`[48], `redux-thunk`[49], and `redux-logger`[50].

Summary

Middleware are an exceptionally versatile and useful part of Redux and are commonly used to hold the most complicated and generic parts of an application. They have access to the current action, to the store, and to the `dispatch()` method, giving them more power than any other part of Redux.

[48] https://github.com/markdalgleish/redux-analytics
[49] https://github.com/gaearon/redux-thunk
[50] https://github.com/evgenyrodionov/redux-logger

Part 3. Redux in the Real World

Chapter 7. State Management

One of the main strengths of Redux is the separation of state (data) management from the presentation and logic layers. This division of responsibilities means the design of the state layout can be done separately from the design of the UI and any complex logic flows.

To illustrate this concept of separation, let's consider our recipe book application. The app can manage multiple recipe books, each having multiple recipes. A recipe, in turn, is an object containing a list of ingredients, preparation instructions, images, a favorited flag, etc.:

Naive structure of recipe app state

```
const state = {
  books: [
    {
      id: 21,
      name: 'Breakfast',
      recipes: [
        {
          id: 63,
          name: 'Omelette',
          favorite: true,
          preparation: 'How to prepare...',
          ingredients: [...]
        },
        {...},
        {...}
      ]
    },
    {...},
    {...}
  ]
};
```

This state layout contains all the required information and conforms exactly to the description of our application, but it requires complex nested reducers and access to deep nested data is complicated. While still workable, these problems lead to convoluted code. Let's explore both of them in detail.

Reducer Nesting and Coupling

Let's try to implement a reducer that supports an action that adds a new ingredient to a recipe. There are two main approaches, one where all the reducers in the chain are aware of the action being passed and one where each reducer only passes the information down to its children.

The first approach could be implemented as follows:

Action-aware reducers

```
const booksReducer = (state, action) => {
  switch (action.type) {
    case ADD_INGREDIENT:
      return Object.assign({}, state, {
        books: state.books.map(
          book => book.id !== action.payload.bookId
            ? book
            : recipesReducer(book, action)
        )
      });
  }
};

const recipesReducer = (book, action) => {
  switch (action.type) {
    case ADD_INGREDIENT:
      return Object.assign({}, book, {
        recipes: book.recipes.map(
          recipe => recipe.id !== action.payload.recipeId
            ? recipe
            : ingredientsReducer(recipe, action)
        )
      });
  }
};

const ingredientsReducer = (recipe, action) => {
  // Regular reducer
};
```

In this implementation, all the "parent" reducers must be aware of any actions used in their children. Any changes or additions will require us to check multiple reducers for code changes, thus breaking the encapsulation benefits of multireducer composition and greatly complicating our code.

The second option is for reducers to pass all actions to their children:

Action-passing reducer

```
const booksReducer = (books, action) => {
  const newBooks = handleBookActions(books, action);

  // Apply recipes reducers
  return newBooks.map(book => Object.assign({}, book, {
    recipes: recipesReducer(book.recipes, action)
  }));
};

const recipesReducer = (recipes, action) => {
  const newRecipes = handleRecipeActions(book, action);

  // Apply ingredients reducers
  return newRecipes.map(recipe => Object.assign({}, recipe, {
    ingredients: ingredientsReducer(recipe.ingredients, action)
  }));
};
```

In this implementation, we separate the reducer logic into two parts: the first to allow any child reducers to run and the second to handle the actions for the reducer itself.

While this implementation doesn't require the parent to know about the actions supported by its children, it means we are forced to run a very large number of reducers for each recipe. A single action unrelated to recipes, like `UPDATE_PROFILE`, will run `recipesReducer()` for each recipe, and it in turn will run `ingredientsReducer()` for each of the ingredients.

Access to Multiple Nested Data Entities

Another problem with the nested state approach is retrieving data. If we would like to show all of a user's favorite recipes, we need to scan all the books to find the relevant ones:

Get list of favorite recipes

```
const getFavorites = (state) => {
  const recipes = state.books.map(
    book => book.recipes.filter(recipe => recipe.favorite)
  );

  // Strip all null values
  return recipes.filter(recipe => recipe);
};
```

Also, since this or similar code will be used for the UI, any changes to structure of the state will need to be reflected not just in the reducers, but in the UI as well. This approach breaks the separation of concerns and increases the amount of work required for state structure changes.

State as a Database

A recommended approach to solve the various issues raised above is to treat the application state as a database of entities. Like in a regular table-based database, we will have a "table" for each entity type with a "row" for each entity. The entity `id` will be our "primary key" for each table.

In our example, we will break down nesting to make our state as shallow as possible and express connections using IDs:

Normalized state

```
const state = {
  books: {
    21: {
      id: 21,
      name: 'Breakfast',
      recipes: [63, 78, 221]
    }
  },

  recipes: {
    63: {
      id: 63,
      book: 21,
      name: 'Omelette',
      favorite: true,
      preparation: 'How to prepare...',
      ingredients: [152, 121]
    },
    78: {},
    221: {}
  },

  ingredients: {}
};
```

In this structure, each object has its own key right in the root of our state. Any connections between objects (e.g., ingredients used in a recipe) can be expressed using a regular ordered array of IDs.

Let's examine the implementation of the reducers needed to handle the `ADD_INGREDIENT` action using the new state structure:

Reducers for adding a new ingredient

```
const booksReducer = (books, action) => {
  // Not related to ingredients any more
};

const recipeReducer = (recipe, action) => {
  switch (action.type) {
    case ADD_INGREDIENT:
      return Object.assign({}, recipe, {
        ingredients: [...recipe.ingredients, action.payload.id]
      })
  }

  return recipe;
};

const recipesReducer = (recipes, action) => {
  switch (action.type) {

    case ADD_INGREDIENT:
      return recipes.map(recipe =>
        recipe.id !== action.payload.recipeId
          ? recipe
          : recipeReducer(recipe, action));
  }
};

const ingredientsReducer = (ingredients, action) => {
  switch (action.type) {
    case ADD_INGREDIENT:
      return [...ingredients, action.payload];
  }
};
```

There are two things to note in this implementation compared to what we saw with the denormalized state:

1. The books reducer is not even mentioned. Nesting levels only affect the parent and children, never the grandparents.
2. The recipes reducer only adds an ID to the array of ingredients, not the whole ingredient object.

To take this example further, the implementation of `UPDATE_RECIPE` would not even require any change to the recipes reducer, as it can be wholly handled by the ingredients reducer.

Access to Multiple Nested Data Entities

Getting a list of favorite recipes is also much simpler with normalized state, as we only need to scan the recipes "table." This can be done in the UI using a function called a *selector*. If you think of the state as a database, you can imagine selectors as database queries:

Favorite recipes selector

```
import filter from 'lodash/filter';

const getFavorites = (state) =>
  filter(state.recipes, 'favorite');
```

The main improvement is that we do not need to be aware of the structure or nesting of the state to access deeply nested information. Rather, we treat our state as a conventional database from which to extract information for the UI.

A more common example might be displaying a single recipe. With the table approach, we can get to any entity directly by its ID:

Getting an entity from the state

```
const getRecipe = (state, id) => state.recipes[id];
```

No deep searches are needed; the data is simple and quick to access.

Keeping a Normalized State

While normalized state might seem like a great idea, often the data returned from the server is structured in a deeply nested way. A possible example of fetching data from the `/recipes/63` API endpoint might look like this:

Data returned from /recipes/63

```
{
  id: 63,
  name: 'Omelette',
  favorite: true,
  preparation: 'How to prepare...',
  ingredients: [
    {
      id: 5123,
      name: 'Egg',
      quantity: 2
    },
    {
      id: 729,
      name: 'Milk',
      quantity: '2 cups'
    }
  ]
};
```

Since the only way to update the Redux store is by sending actions to the reducers, we must build a payload that can be easily handled by our reducers and find a way to extract the payload from the denormalized data returned from the server.

Building the Generic Action

Potentially, we would like each of our data reducers to be able to handle a special `UPDATE_DATA` action and extract the relevant parts it needs:

UPDATE_DATA action

```
const updateData = ({
  type: UPDATE_DATA,
  payload: {
    recipes: {
      63: {
        id: 63,
        name: 'Omelette',
        favorite: true,
        preparation: 'How to prepare...',
        ingredients: [5123, 729]
      }
    },
    ingredients: {
      5123: {
        id: 5123,
        name: 'Egg',
        quantity: 2
      },
      729: {
        id: 729,
        name: 'Milk',
        quantity: '2 cups'
      }
    }
  }
});
```

Using this approach, our recipes reducer's support for UPDATE_DATA can be as simple as:

Recipes reducer support for UPDATE_DATA

```
const recipesReducer = (state, action) => {
  switch (action.type) {
    case UPDATE_DATA:
      if (!('recipes' in action.payload)) return state;

      return Object.assign({}, state, {
        recipes: Object.assign({},
          state.recipes,
          action.payload.recipes
        )
      });
  }
};
```

Our reducer checks if the payload contains any recipes and merges the new data with the old recipes object, thus adding to or otherwise modifying it as needed.

Normalizing the Data

With the reducers updated and the action structure defined, we are left with the problem of extracting the payload from the denormalized data we got from the server.

A simple approach might be to have a custom function that knows each API's shape and normalizes the returned nested JSON into a flat structure with custom code.

Since this is quite a common practice, the custom code can be replaced by the normalizr[51] library. That is, we can define the schema of the data coming from the server and have the normalizr code turn our nested JSON into a normalized structure we can pass directly into our new UPDATE_DATA action.

[51]https://github.com/paularmstrong/normalizr

To use the library, we first define the schema for our data:

Defining the schema

```
import { schema } from 'normalizr';

export const ingredient = new schema.Entity('ingredient');

export const recipe = new schema.Entity('recipe', {
  ingredients: [ingredient]
});
```

Here we defined two entity types, a basic `ingredient` and a `recipe` that holds an *array* of `ingredient` objects.

The `normalizr` library allows for much more complex schemas. Consult the documentation for details.

Once we have obtained the data from the server and defined a schema for the data, we can use `normalizr` to automatically normalize the data:

Normalizing data returned from the server

```
import { normalize } from 'normalizr';

const data = normalize(data, recipe);
```

The `data` object will now hold two keys:

1. `entities`—An object containing the normalized entities, arranged by entity type
2. `result`—The ID of the main object or array (if we passed multiple recipes as data)

Inside the `entities` key we will find the normalized data:

Data after normalization

```
{
  ingredients: {
    5123: {
      id: 5123,
      name: 'Egg',
      quantity: 2
    },
    729: {
      id: 729,
      name: 'Milk',
      quantity: '2 cups'
    }
  },
  recipes: {
    63: {
      id: 63,
      name: 'Omelette',
      favorite: true,
      preparation: 'How to prepare...',
      ingredients: [5123, 729]
    }
  }
}
```

The `normalizr` library did all the hard work for us. Now we can update our store by sending the relevant parts to each reducer:

Dispatching two actions to update state

```
dispatch({ type: 'SET_INGREDIENTS', payload: data.entities.ingredients });
dispatch({ type: 'SET_RECIPES', payload: data.entities.recipes});
```

With this approach each reducer only has to handle its own entity types, making the reducer code simple and nondependent on other actions and entities.

It is best to keep the schemas of all the entities in a separate (and reusable) place in your application; for example, in a *lib/schema.js* file.

Here's a complete example of the use of this handy library:

Complete normalizr example

```
import { normalize, schema } from 'normalizr';

export const ingredient = new schema.Entity('ingredient');

export const recipe = new schema.Entity('recipe', {
  ingredients: [ingredient]
});

const handleRecipeData = (data, dispatch) => {
  data = normalize(data, recipe);

  dispatch({ type: 'SET_INGREDIENTS', payload: data.entities.ingredients });
  dispatch({ type: 'SET_RECIPES', payload: data.entities.recipes });
};
```

Persisting State

In many cases we will want to keep the current state after a page refresh or the application's tab being closed. The simplest approach to persisting the state is keeping it in the browser's `localStorage`.

To easily sync our store with `localStorage` (or any other storage engine), we can use the `redux-persist`[52] library. This will automatically serialize and save our state once it has been modified.

To use the library, we need to install it with `npm` (`npm install -s redux-persist`) and modify the store and reducer creation files to add the functionality.

[52] https://github.com/rt2zz/redux-persist

Updating the Root Reducer

First, we update the root reducer to support persistence:

Original reducers/root.js

```
import { combineReducers } from 'redux';
import recipes from './recipes';
import ingredients from './ingredients';

export default combineReducers({
  recipes,
  ingredients
});
```

Adding persistence to reducers/root.js

```
import { persistCombineReducers } from 'redux-persist'
import recipes from './recipes';
import ingredients from './ingredients';
import storage from 'redux-persist/es/storage';

const config = { key: 'root', storage };

export default persistCombineReducers(config, {
  recipes,
  ingredients
});
```

In the new code, we are using the `persistCombineReducers()` method of `redux-persist` instead of the regular `combineReducers()` from `redux`. An additional `config` parameter provides the information needed by `redux-persist` on where and how to store the state.

The `config` object has many other configuration fields, described in detail in the `redux-persist` library's documentation.

Updating the Store

Next, we add support for persistence to the store:

Original store.js

```
import { createStore } from 'redux';
import reducer from 'reducers/root';

const store = createStore(reducer);

export default store;
```

Adding persistence to store.js

```
import { createStore } from 'redux';
import { persistStore } from 'redux-persist';
import reducer from 'reducers/root';

const store = createStore(reducer);
const persistor = persistStore(store);

export default { store, persistor };
```

To have our store support persistence, we create a new `persistor` object wrapping our original store. While we are still going to use the regular `store` in our application, the new `persistor` object allows for more advanced functionality (e.g., smarter updates to the UI).

With both of these changes done, our store and the browser's `localStorage` will automatically be in sync and our state will persist across page refreshes.

Advanced State Sync

The `redux-persist` library has advanced functionality that will allow us to whitelist only part of the state to persist and to specify special serializers for the part of the state that cannot be serialized with `JSON.stringify()` (functions, symbols, etc.). We recommend that you review the library's documentation for details on the more advanced features.

Real-World State

In a real-world application, our state will usually contain a number of different entities, including the application data itself (preferably normalized) and auxiliary data (e.g., the current access token, or pending notifications).

Common State Structure

Unlike the data coming from the server, some information will be used exclusively by our front-end application for its internal needs. A common example would be keeping the total number of active server requests in order to know whether or not to display a spinner. Or we might have a `currentUser` property where we keep information about the current user, such as their username and access token:

Sample state

```
const state = {
  books: { },
  recipes: { },
  ingredients: { },
  ui: {
    activeRequests: 0
  },
  currentUser: {
    name: 'Kipi',
    accessToken: 'topsecrettoken'
  }
};
```

As our application grows, more types of state entities will creep in. Some of these will come from external libraries like `redux-forms` or `react-router-redux`, or others that require their own place in our state. Other entities will come from the application's business needs.

For example, if we need to support editing of the user's profile with the option to cancel, our implementation might create a new `temp` key where we will store a copy of the profile while it is being edited. Once the user clicks "confirm" or "cancel," the temporary copy will either be copied over to become the new profile or simply deleted.

Keeping the State Manageable

To make things as simple as possible, it is best to have a reducer for each key in the state. This will allow for an encapsulation approach where it is immediately clear where different parts of our state can be modified.

For a very large project, it might be beneficial to separate the "server data" and the "auxiliary/temporary data" under different root keys:

Large nested state

```
const state = {
  db: {
    books: { },
    recipes: { },
    ingredients: { }
  },
  local: {
    ui: {
      activeRequests: 0
    },
    user: {
      name: 'Kipi',
      accessToken: 'topsecrettoken'
    }
  },
  vendor: {
    forms: {},
    router: {}
  }
};
```

This allows for easier management of different parts when deciding what has to be synced with `localStorage`, or when clearing stale data.

In general, the state is the front end's database and should be treated as such. It is important to periodically check the current layout and do any refactoring that's necessary to make sure the state's structure is clean, clear, and easy to extend.

What to Put in the State

A common issue when working with Redux is deciding what information goes inside our state and what is left outside, either in React's state, in Angular's services, or in other storage methods of different UI libraries.

There are a few questions to consider when deciding whether to add something to the state:

- Should this data be persisted between page refreshes?
- Should this data be persisted across route changes?
- Is this data used in multiple places in the UI?

If the answer to any of these questions is “yes,” the data should go into the state. If the answer to all of these questions is “no,” if you’re looking to reduce the complexity of your state, keep it out.

A few examples of data that can be kept outside of the state are:

- Currently selected tab in a tab control on a page
- Hover visibility/invisibility on a control
- Lightbox being open/closed
- Currently displayed errors

You can think of this as similar to putting data in a database or keeping it temporarily in memory. Some information can be safely lost without affecting the user’s experience or corrupting their data.

Summary

In this chapter we discussed the structure of our Redux state and how it should be managed for easier integration with reducers and the UI. We also learned that the state should be considered the application’s database and be designed separately from the presentation and logic layers.

In the next chapter we will talk about server communication, the best method of sending data to and receiving it from our server—using middleware.

Chapter 8. Server Communication

Server communication is vital for most web applications. The basics can be implemented with a few lines of code, but it can quickly grow into a complicated mechanism that handles authentication, caching, error handling, WebSockets, and a myriad of other features and edge cases.

For simple applications, the most commonly taught approaches are either writing the server communication code inside the UI layer or using libraries like `redux-thunk`[53] that allow action creators to dispatch actions asynchronously.

When a project grows, however, it becomes clear that it's best to have a single place to handle authentication (setting custom headers), errors, and features like caching. This means we need to be able to access the store and dispatch asynchronous events—which is the perfect job for a middleware.

There are numerous solutions for performing actual network calls, from abstract third-party libraries like `superagent`[54], `request`[55], and `axios`[56] to the native low-level `XMLHttpRequest`[57] and `fetch()`[58] APIs. Each has its pros and cons, and comparing them is out of scope for this book. In our examples we will use `axios` for simplicity.

If you are not familiar with middleware, it is strongly recommended that you read the Middleware chapter before proceeding.

[53] https://github.com/gaearon/redux-thunk
[54] https://github.com/visionmedia/superagent
[55] https://github.com/request/request
[56] https://github.com/axios/axios
[57] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[58] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Asynchronous Action Creators

Let's start by looking at a server communication implementation with async action creators using the `redux-thunk` library to understand the underlying pitfalls of this approach:

Promise in action creator

```
const fetchUser = id => dispatch =>
  axios.get(`http://api.ourserver.com/user/${id}`)
    .then(({ data: userData }) => dispatch(setUserData(userData)));
```

Consider another example where we would like to fetch a list of comments for a user:

Promise in action creator

```
const fetchComments = id => dispatch =>
  axios.get(`http://api.ourserver.com/user/${id}/comments`)
    .then(({ data: commentsData }) => dispatch(setComments(commentsData)));
```

Even without the more complex functionality mentioned earlier, the code contains a lot of duplication between action creators. And there are a few other problematic issues:

- We can't log actions before the network request completes, preventing the regular Redux debug flow.
- Every action creator has repetitive functionality for handling errors and setting headers.
- Testing gets harder as more async action creators are added.
- If we want to change the server communication strategy (e.g., replace `axios` with `superagent`), we need to change multiple places in the code base.

Keeping the code as short and stateless as possible is easier when all the action creators are simple functions. This makes them easier to understand, debug, and test. But keeping the action creators clean from async code means moving that code into another part of the stack. Luckily, we have the perfect candidate—middleware. As we will see, using this approach we can keep action creators simple and generate actions that contain all the information needed for a middleware to perform the API requests.

API Middleware

A quick glance at the previous two action creator examples reveals that there are only two differences between the functions: the URL and the action that gets dispatched when data is successfully fetched.

In the spirit of generality, we can create an abstract layer to handle network requests and only pass it the `url` and `onSuccess` parameters. But using a shared function or library does not fully solve all the problems mentioned earlier. Our action creators will still be asynchronous, which means they'll be harder to test and more difficult to debug using the regular Redux debug flow of monitoring all actions dispatched to the store.

Another approach would be to create a new middleware that will handle all the server communication:

Simple API middleware

```
const apiMiddleware = ({ dispatch }) => next => action => {
  next(action);

  if (action.type !== 'API') {
   return;
  }

  // Handle network requests
};
```

Our middleware will listen to any action with type `'API'` and use the information in its payload to make a request to the server. It will let any other actions flow down the middleware chain.

Moving Code from Action Creators

We can now take the examples from the beginning of this chapter and encode the differences in the action's payload:

Before: asynchronous action creators

```
const fetchUser = id => dispatch =>
  axios.get(`http://api.ourserver.com/user/${id}`)
    .then(({ data: userData }) => dispatch(setUserData(userData)));

const fetchComments = id => dispatch =>
  axios.get(`http://api.ourserver.com/user/${id}/comments`)
    .then(({ data: commentsData }) => dispatch(setComments(commentsData)));
```

After: action creators that return plain objects

```
const fetchUser = id => ({
  type: API,
  payload: {
    url: `http://api.ourserver.com/user/${id}`,
    onSuccess: setUserData
  }
});

const fetchComments = id => ({
  type: API,
  payload: {
    url: `http://api.ourserver.com/user/${id}/comments`,
    onSuccess: setComments
  }
});
```

The new action creators return plain JavaScript objects that are easy to test. Now we can move all the asynchronous code into our new API middleware:

Simple API middleware

```
const apiMiddleware = ({ dispatch }) => next => action => {
  next(action);

  if (action.type !== API) {
    return;
  }

  const { url, onSuccess } = action.payload;

  axios.get(url)
    .then(({ data }) => dispatch(onSuccess(data)));
};
```

It should immediately become clear that the middleware approach will give us a single place to encapsulate all the requirements of a server communication layer. One of the biggest benefits is that all our action creators will now return plain objects instead of async functions. This makes action creators much easier to create and test. With the API middleware approach, server communication is now declarative: we simply build an object describing the required network call.

Avoiding Full URLs

Passing full URLs from each action creator might cause code bloat and make it harder to run in different environments (development, staging, test, or production). Action creators should only pass the relative path of the URL, and API middleware should combine it with a constant set at build time:

Using a predefined base URL

```
axios.get(process.env.BASE_URL + action.payload.url);

// Alternatively, create an axios client with a predefined base URL
// const client = axios.create({ baseURL: process.env.BASE_URL });
```

Before extracting base URL

```
const fetchUser = id => ({
  type: API,
  payload: {
    url: `http://api.ourserver.com/user/${id}`,
    onSuccess: setUserData
  }
});

const fetchComments = id => ({
  type: API,
  payload: {
    url: `http://api.ourserver.com/user/${id}/comments`,
    onSuccess: setComments
  }
});
```

After extracting base URL

```
const fetchUser = id => ({
  type: API,
  payload: {
    url: `/user/${id}`,
    onSuccess: setUserData
  }
});

const fetchComments = id => ({
  type: API,
  payload: {
    url: `/user/${id}/comments`,
    onSuccess: setComments
  }
});
```

In this chapter we will be using just the `url` parameter, assuming that `BASE_URL` is already defined for us.

Passing Functions or Strings

A common dilemma when using the API middleware approach is deciding whether the `onSuccess` parameter should be an action creator:

Passing an action creator

```
import { setUserData } from '../actions/users';

const fetchUser = id => ({
  type: API,
  payload: {
    url: `/user/${id}`,
    onSuccess: setUserData
  }
});
```

or a type constant:

Passing an action type

```
import { SET_USER_DATA } from '../constants/actionTypes';

const fetchUser = id => ({
  type: API,
  payload: {
    url: `/user/${id}`,
    onSuccess: SET_USER_DATA
  }
});
```

The difference between these approaches is in the way the API middleware dispatches an action once a network request is resolved:

Receiving an action creator

```
axios.get(url)
   .then(({ data }) => dispatch(onSuccess(data)));
```

Receiving an action type

```
axios.get(url)
   .then(({ data }) => dispatch({ type: onSuccess, payload:  data }));
```

Both approaches have their pros and cons. Passing an action type makes actions serializable and thus easier to output to logs and debug visually. However, because this approach separates actions from the results of network calls, it might be harder to follow the flow of side effects created by the actions.

Error Handling

Our current example ignores error handling. To solve this problem, we need to extend our middleware to catch server errors and dispatch events when they happen:

Handling API errors

```
axios.get(url)
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => dispatch(apiError(error)));
```

The dispatched action creator `apiError(error)` could be caught by a special reducer to display an error message or another middleware for more processing.

In more complex scenarios, we might need an API request to have a custom error handler. To support this, we can extend our API middleware to support not just the `onSuccess` parameter in the action creator, but also an `onFailure` one:

Passing a custom error action creator

```
const fetchUser = id => ({
  type: API,
  payload: {
    url: `/user/${id}`,
    onSuccess: setUserData,
    onFailure: raiseAlarm
  }
});
```

We can now extend our API middleware to dispatch the `action.payload.onFailure()` action creator every time we have a failed network request or receive an error from the server:

Handling custom API errors

```
const { url, onSuccess, onFailure } = action.payload;

axios.get(url)
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => {
    dispatch(apiError(error));

    if (onFailure) {
      dispatch(onFailure(error));
    }
  });
```

Handling Unauthorized Requests

A common error returned from the server is `401 Unauthorized`. This usually happens when users try to access a part of the system they don't have permissions to access, or when their credentials have expired. We can easily extend our API middleware to automatically catch authorization errors and cause a generic reaction. The simplest approach in these cases is to log the users out and ask them to reauthenticate in order to (potentially) gain access to a previously inaccessible area of the application.

To do so, we will need to create a new `logOut()` action creator and implement the flow in other parts of our stack. This usually means clearing all data, removing the access token, and redirecting the user to the login screen:

Handling unauthorized requests

```
import { logOut } from '../actions/auth';
const { url, onSuccess, onFailure } = action.payload;

axios.get(url)
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => {
    dispatch(apiError(error));

    if (onFailure) {
      dispatch(onFailure(error));
    }

    if (error.response.status === 401) {
      // Remember which URL failed and try to go back after login
      dispatch(logOut(window.location.pathname));
    }
  });
```

The preceding code uses another trick to pass the current URL to the `logOut()` action creator. This will allow us to save the current URL before navigating to the login page and potentially bring the user back to it after a successful login.

While this approach might work at the beginning, our error handling code will become more complex and application-specific as the code size grows. A cleaner approach would be to have the API middleware only emit an error action and use a different middleware to catch and handle various errors:

Splitting error handling

```
import { accessDenied } from '../actions/auth';
const { url, onSuccess, onFailure } = action.payload;

axios.get(url)
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => {
    dispatch(apiError(error));

    if (onFailure) {
      dispatch(onFailure(error));
    }

    if (error.response.status === 401) {
      dispatch(accessDenied());
    }
  });
```

Now the API middleware only sends the `accessDenied()` action, and we create different middleware to handle the (complex) flow of handling the problem:

Handling authentication errors

```
import { ACCESS_DENIED } from '../consts/action-types';
import { logOut } from '../actions/auth';

const authMiddleware = ({ dispatch }) => next => action => {
  next(action);

  if (action.type === ACCESS_DENIED) {
    // Remember which URL failed and try to go back after login
    dispatch(logOut(window.location.pathname));
  }
};
```

Authentication

A common place to store the current user's information (such as the access token) is in the Redux store. As all our API logic is now located in one place and the middleware has full access to the store using the `getState()` method, we can extract the `accessToken` from the state and set it as a header for our server requests:

Setting the access token as a header

```
const defaultHeaders = {};
const { accessToken } = getState().currentUser;

if (accessToken) {
  Object.assign(defaultHeaders, { 'Authorization': `Bearer ${accessToken}` });
}

axios.get(url, { headers })
```

All our server calls will now automatically get the correct headers without us having to worry about this in any other parts of our application.

Handling Other HTTP Methods

Server communication is not only about fetching data. We also need to be able to create, update, and delete resources. We can extend our API middleware to accept a new `method` property in the action payload to differentiate between request types:

Updating the user's name

```
const updateUserName = (id, name) => ({
  type: API,
  payload: {
    url: `/user/${id}`,
    method: 'POST',
    data: { name },
    onSuccess: setUserData
  }
});
```

This new action creator will build an action object containing two new features, the `method` to use and the `data` to send to the server:

Handling non-GET requests

```
const { url, onSuccess, onFailure, method = 'GET', data } = action.payload;

axios.request({ url, method, data })
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => { /* handle errors */ });
```

Passing Query Parameters in GET Requests

In some cases we might want to pass data to a `GET` request. Most libraries have their own approaches to serialization of parameters to query strings suitable for `GET` requests. In the case of the `axios` library, it requires a different property on the request configuration object to be sent with query parameters. We can easily determine that using the parameters passed in the action's payload:

Passing query params in GET requests

```
const { url, onSuccess, onFailure, method = 'GET', data } = action.payload;
const dataProperty = ['GET', 'DELETE'].includes(method) ? 'params' : 'data';

axios.request({ url, method, [dataProperty]: data })
  .then(({ data }) => dispatch(onSuccess(data)))
  .catch(error => { /* handle errors */ });
```

Implementing a Loading Indicator (Spinner)

A common question when using Redux is how to show a spinner when server requests are in progress. The middleware approach provides an answer using additional `dispatch()` calls. Before a request starts or after it completes, we can dispatch a special action to be caught by reducers responsible for the UI state.

Here, we dispatch an `apiStart()` action before starting any server communication and dispatch `apiFinish()` on both successful and failed server responses:

Showing and hiding a spinner

```
dispatch(apiStart());

axios.request({ url, method })
  .then(({ data }) => {
    // handle response
    dispatch(apiFinish());
  })
  .catch(error => {
    // handle errors
    dispatch(apiFinish());
  });
```

To keep track of pending requests, we can keep a counter in a `pendingRequests` or `ui` container in the state. The counter can be used by the UI to show a spinner if the number of pending requests is greater than zero:

UI reducer to handle the requests counter

```
const uiReducer = (state, action) => {
  switch (action.type) {
    case API_START:
      return Object.assign({}, state, {
        pendingRequests: state.pendingRequests + 1
      });

    case API_FINISH:
      return Object.assign({}, state, {
        pendingRequests: state.pendingRequests - 1
      });
    }
};
```

Our state will now contain an up-to-date count of active server requests in `state.ui.pendingRequests`, which can be easily used to display a spinner or similar UX effect to the user. One caveat with this feature is that sometimes you will have to manually reset the number of pending requests when cancelling requests or navigating between pages to hide the spinner for stale requests.

Multiple Spinners

In single-page applications, the better UX practice is usually not to block the entire UI with one loading indicator, but to show multiple localized indicators depending on different pending network requests. An example could be two independent loading indicators for movie details and reviews.

A simple solution to this problem would be adding a `label` property to our API action payload and having the API middleware pass it to `apiStart()` and `apiFinish()` actions:

Movies and reviews action creators

```
const fetchMovie = (movieId) => ({
  type: API,
  payload: {
    url: `/movie/${movieId}`,
    onSuccess: setMovie,
    label: 'movie'
  }
});

const fetchComments = (movieId) => ({
  type: API,
  payload: {
    url: `/movie/${movieId}`,
    onSuccess: setReviews,
    label: 'reviews'
  }
});
```

Label-based network counters

```
dispatch(apiStart(label));

axios.request({ url, method })
  .then(({ data }) => {
    // handle response
    dispatch(apiFinish(label));
  })
  .catch(error => {
    // handle errors
    dispatch(apiFinish(label));
  });
```

Transforming Data

In many cases servers do not return data in the same structure in which we organize it in our state. This means that some API endpoints might require special preprocessing of data before it is sent to the server, or special post-processing of the response before it is stored in the state.

The API middleware offers a simple approach to the problem by adding support for `transformRequest` and `transformResponse` properties to the action object:

Action creator that handles "bad" APIs

```
const fixWeirdCommentsAPI = data => {
  // Do some conversion of data
  return data.results.hidden[0].actualData;
};

const fetchComments = () => ({
  type: API,
  payload: {
    url: '/comments',
    onSuccess: setComments,
    transformResponse: fixWeirdCommentsAPI
  }
});
```

We will usually store a function like `fixWeirdCommentsAPI()` in a special library and test it separately (with the hope of removing it in the brighter future).

Libraries like `axios` sometimes provide request and response transformations as part of their APIs, which makes supporting this feature virtually effortless:

Action creator that handles "bad" APIs

```
const {
  url,
  onSuccess,
  onFailure,
  method = 'GET',
  data,
  transformRequest,
  transformResponse
} = action.payload;

const dataProperty = ['GET', 'DELETE'].includes(method) ? 'params' : 'data';

axios.request({
  url,
  method,
  [dataProperty]: data,
  transformRequest,
  transformResponse
})
  .then(response => dispatch(onSuccess(response.data)))
  .catch(error => { /* handle errors */ });
```

However, if you use low-level browser APIs to make requests, adding data processing is still relatively easy:

Action creator that handles "bad" APIs

```
const { url, onSuccess, transformResponse } = action.payload;

axios.get(url)
  .then(({ data }) => dispatch(
    onSuccess(transformResponse ? transformResponse(data) : data)
  ));
```

Normalizing Responses

In Redux applications where the `normalizr`[59] library is used, the API middleware can automatically apply the normalization on the received data:

Auto-normalization

```
import { normalize } from 'normalizr';

const {
  url,
  onSuccess,
  onFailure,
  method = 'GET',
  data,
  transformRequest,
  transformResponse,
  schema
} = action.payload;

const dataProperty = ['GET', 'DELETE'].includes(method) ? 'params' : 'data';

axios.request({
  url,
  method,
  [dataProperty]: data,
  transformRequest,
  transformResponse
})
  .then(({ data }) => dispatch(onSuccess(normalize(data, schema))))
  .catch(error => { /* handle errors */ });
```

More information about `normalizr` can be found in the State Management chapter.

Our action creators can define a schema to automatically convert a complex nested response. However, normalized responses usually consist of data about multiple entities and require either handling one response action in multiple reducers or, better, sending multiple actions to update different parts of the state according to the response from the server.

[59] https://github.com/paularmstrong/normalizr

For example, this action creator defines a network call that will automatically normalize a complex response containing an array of movies, each having a list of reviews and actors:

Passing a normalizr schema in an action creator

```
import { moviesSchema } from '../schemas';

const fetchMovies = () => ({
  type: API,
  payload: {
    url: '/movies',
    schema: moviesSchema,
    onSuccess: ({ entities, result }) => [
      setMovies(result),
      setAuthors(entities.authors),
      setReviews(entities.reviews)
    ]
  }
});
```

You might notice that the `onSuccess` property now contains a function that returns an array of actions instead of a single action creator. While we can easily add support for this functionality in our API middleware, a more generic solution would be to add a new middleware to the stack. This new middleware will catch all actions that are of type `Array` and dispatch them one at a time:

Middleware to handle arrays of actions

```
const actionArrayMiddleware = ({ dispatch }) => next => action => {
  if (Array.isArray(action)) {
    action.forEach(dispatch);
    return;
  }

   next(action);
};

export default actionArrayMiddleware;
```

Chaining Network Requests

Sometimes in order to fetch data from the server we have to make chained API calls. A simple example might be fetching the metadata of the current user and then the user's actual profile from different endpoints. Here's an example using an asynchronous action creator with `redux-thunk`:

Chaining promises

```
const fetchCurrentUser = () => dispatch =>
 axios.get(`/user`)
  .then(({ data: user }) => {
    dispatch(setUser(user));

    // Get user's profile
    axios.get(`/profile/${user.profileId}`)
      .then(({ data: profile }) => dispatch(setProfile(profile)));
    }
  );
```

The solution appears to be quite straightforward, but there are a few issues with this code:

- Exact branching and catching of errors is not obvious and can become complex as the chain grows.
- It is hard to debug and understand what stage in the chain we are currently at.
- It is nearly impossible to cancel or abort the chain if the user navigates to a different part of the UI and the current request chain is no longer needed.

Clearly, chaining promises is not an ideal solution. Ultimately what we need to do is to dispatch additional actions as a result of the previous action's success. This specific case is called a *side effect*, and it is part of a broader concept we will discuss in the Managing Side Effects chapter.

There are a lot of ways to achieve the same goal, but we are going to concentrate on just a few. With any approach, our API middleware should remain the same and only handle the network communication; the only thing that will be different is where and how we organize our flow logic.

Let's first take a look at the approach we saw earlier, using `actionArrayMiddleware`. With this approach, we define the next actions as part of a success callback passed within the action's payload.

Note that the action still remains a plain JavaScript object (although not serializable) and there is no logic or async code involved within the action itself:

Chaining action creators

```
const fetchCurrentUser = () => ({
  type: API,
  payload: {
    url: '/user',
    onSuccess: (user) => [
      setUser(user),
      fetchProfile(user.profileId)
    ]
  }
});

const fetchProfile = (profileId) => ({
  type: API,
  payload: {
    url: `/profile/${profileId}`,
    onSuccess: setProfile
  }
});
```

This approach keeps all the server communication logic hidden in the API middleware and exposes only the logic interface—but there are some downsides too. First of all, the whole flow is distributed between multiple functions (if not files), and the deeper the dependencies are the more time it will take to understand the flow. Then, there is no way to update the store only after all network calls are resolved. And since the actions are not serializable, we can't use them together with workers or sockets.

Another approach is adding an additional middleware that will handle the complex flow. It will listen to the action(s) resulting from an API call and dispatch additional network requests as needed. This topic is covered in more detail in the next chapter.

Summary

In this chapter we have learned about various approaches to setting up a comprehensive mechanism for server communication. We have used Redux's concept of middleware to move most of the complicated and asynchronous logic away from our action creators and created a single place where error handling, caching, and other aspects of network calls can be concentrated.

In the next chapter we will dive deeper into managing side effects beyond chaining network requests.

Chapter 9. Managing Side Effects

Redux is a great tool that addresses one of the main problems of UI frameworks: state management. The unidirectional data flow model makes it easy to understand how events change the state. However, there's one problem Redux doesn't solve out of the box: the management of side effects.

Actions are descriptions of events originated by a user, a network call, or a timer. In some cases a single action might generate multiple other actions in a synchronous or asynchronous manner. One example is executing a number of actions after a user has logged in, such as fetching user information, showing new notifications, and maybe opening a tutorial.

Since Redux does not provide a solution out of the box, the community has created a number of libraries to tackle this problem: `redux-observable`[60], `redux-loop`[61], `redux-saga`[62], `redux-promise`[63], `redux-effects`[64], `redux-thunk`[65], and more.

Each of these solutions is built with a different approach and mental model in mind and has its own pros and cons. Some of them suggest managing side effects in the UI itself, which we find a bad practice since we want the UI layer to be completely disconnected from business logic. Some require additional libraries. Some suggest describing flows in action creators. Most of them, though, rely on the middleware part of the Redux stack to handle the side effects.

We will skip the long conversation about handling business logic flows in the UI layer by saying *just don't do it* and proceed straight to discussing other approaches.

---

[60]https://redux-observable.js.org
[61]https://github.com/redux-loop/redux-loop
[62]https://redux-saga.js.org
[63]https://github.com/redux-utilities/redux-promise
[64]https://github.com/redux-effects/redux-effects
[65]https://github.com/gaearon/redux-thunk

Side Effects in Action Creators

One of the most common approaches seen in online Redux tutorials is to handle side effects inside action creators using `redux-thunk`. The solution appears to be quite straightforward:

Handling side effects with redux-thunk

```
const addItemToCart = item => (dispatch, getState) => {
  dispatch(updateCart(item));

  if (getState().cart.items.length > 10) {
    dispatch(applyDiscount(20));
  }

  dispatch(showNotification(`${item.name} has been added to cart!`));
}
```

If you are building a very small application for yourself without any thought for future scalability, using `redux-thunk` might be the best and easiest option. However, if you are working on a large application either alone or with a team, we would strongly advise against using `redux-thunk` to describe flows.

Besides the issues with asynchronous thunk action creators described in the Server Communication chapter, the biggest problem with handling side effects with `redux-thunk` is the lack of connection between the actual function and the side effects. Instead of triggering one action, a click on a button might trigger three different actions, and without conscious effort to add logging it might be very hard to understand the causality and origin of side effects.

Side Effects in Middleware

Let's take a moment and implement the exact same flow, this time in a middleware:

Handling side effects in middleware

```
const cartMiddleware = ({ dispatch, getState }) => next => action => {
  next(action);

  if (action.type === ADD_ITEM_TO_CART) {
    dispatch(updateCart(item));

    if (getState().cart.items.length > 10) {
      dispatch(applyDiscount(20));
    }

    dispatch(showNotification(`${item.name} has been added to cart!`));
  }
};
```

At first glance it looks practically the same; some might even think there is quite a bit of boilerplate code involved. However, this approach has at least one big advantage over the `redux-thunk` example: the side effects are triggered *as a result of another action*, not as a result of the user's interaction with the UI. This allows for easier debugging, since we can trace all actions in the flow and clearly see what actions are sent as a result of others.

Besides that, you might want to create separate middleware to handle domain-specific parts of the business logic. For instance, if you have additional business rules for discounts, you might want to create a special `discountMiddleware` to handle all the cases. Or, if you have specific requirements for notifications, you could combine all logic for notifications in a `notificationsMiddleware`.

In large applications you would have a set of middleware dedicated to each big feature, just as with actions and reducers. The downsides of this approach are a bit more code and a steeper learning curve for new developers being introduced to the code base; but structured in the right way, middleware scales much better than logic in action creators.

Other Solutions

There are numerous solutions for handling side effects maintained by the Redux community, so why reinvent the wheel? We would encourage you to do your research and play with different approaches before you commit to any of them, however. The larger the project, the harder and more stressful it is to change an existing approach to managing side effects.

Every third-party library has its learning curve and scalability potential. While `redux-thunk` and `redux-promise` might be very easy to learn, they become painful to use as a project grows. Conversely, libraries like `redux-observable` or `redux-saga` scale better but require learning new concepts, like sagas and RxJS (which may or may not be a problem, depending on the prior knowledge of your team and the level of experience of the engineers working on the project).

In this book we concentrate on using middleware to handle side effects because of the benefits it offers: it's closer to the existing tooling available in Redux, supports most use cases, and does not require additional libraries.

If you decide to try going in the lean middleware direction, it might be useful to get your team familiar with the common design patterns described in the next section. You might even be unconsciously using them in your applications already, since they are obvious and logical ways to organize events in a system.

Messaging Patterns

At its core, Redux is all about actions that change the state. Although its architecture wasn't directly inspired by the concepts of event sourcing and the command-query separation principle (which was invented over three decades ago), the fundamentals are close and allow us to use existing principles and patterns to handle side effects. There are two main types of messaging patterns to consider when working with Redux: routing patterns and transformation patterns.

You can find more information on the patterns described here and others in the great book *Enterprise Integration Patterns*[66] by Gregor Hohpe and Bobby Woolf.

Routing Patterns

Technically speaking, routing patterns are used to decouple a message's source from the ultimate destination of the message. To simplify, we could also say that these patterns are used to decide what side effects are triggered in response to actions. Let's take a look at some common routing patterns: filtering, mapping, splitting, and aggregation.

[66] https://en.wikipedia.org/wiki/Enterprise_Integration_Patterns

Filter

Filtering is useful when you have some actions, but have to dispose of some of them based on certain criteria. Examples include debouncing or throttling actions. If you receive Redux messages through a WebSocket every half a second, you might want to throttle them and only update the state once every few seconds to increase performance and the responsiveness of the UI.

Debounce pattern

```
let timeout;

if (action.type === AUTOCOMPLETE) {
  clearTimeout(timeout);
  timeout = setTimeout(() => next(action), 500);
}
```

Mapper

Mapping refers to triggering a different side effect from an action depending on either the content of the action itself or the context of the application. For instance, you might want to handle network calls differently depending on whether the application is online or offline.

Mapper pattern

```
if (action.type === TRACK_PAGEVIEW) {
  if (process.env.NODE_ENV === 'development') {
    return;
  }

  if (process.env.NODE_ENV === 'staging') {
    next({ ...action, accountId: 'UA1-QW122WE' });
  }

  if (process.env.NODE_ENV === 'production') {
    next(action);
  }
}
```

Splitter

This pattern is useful for dispatching multiple actions as a response to another action. For example, calling `ORDER_ITEM` might trigger `API_REQUEST` and `TRACK_ORDER` actions.

Splitter pattern

```
if (action.type === ORDER_ITEM) {
  dispatch({ type: API_REQUEST });
  dispatch({ type: TRACK_ORDER });
}
```

Aggregator

Aggregation refers to triggering a side effect as a result of multiple actions. An example could be sending a `LOCK_ACCOUNT` action after three unsuccessful `LOGIN` actions or sending a `START_GAME` action once two players have joined the game.

Aggregator pattern

```
const players = [];

if (action.type === ADD_PLAYER) {
  players.push(action.payload.player);

  if (players.length === 2) {
    dispatch({ type: START_GAME });
  }
}
```

Transformation Patterns

Transformation patterns (such as enriching, normalization, and translation) change the contents of actions before they reach the reducer. Sometimes an action that is sent from the UI is different from the action that is expected by the reducer or middleware, either because it is missing some of the required information at the time of sending or because the original action is more eloquent in terms of its API.

Enricher

Enriching refers to adding missing properties to an action. For instance, you might want to add the current date to a `SUBMIT_ORDER` action. Instead of making the UI aware of specific business logic and adding the date from there, you could extend the action by adding another property to the action payload.

Enricher pattern

```
if (action.type === SUBMIT_ORDER) {
  next({ ...action, date: new Date() });
}
```

Normalizer

Normalization—where your server returns a different structure from what is used on the client side—is very common in the server communication layer. Usually it is done by changing the server's response before sending another action with the normalized result.

Normalizer pattern

```
if (action.type === FETCH_MOVIES_SUCCESS) {
  dispatch({
    type: SET_MOVIES,
    payload: normalize(action.payload, movieSchema)
  });
}
```

Translator

Sometimes to clarify the UI you might want to dispatch actions that are different from those expected by the reducers. One example is having two separate `SHOW` and `HIDE` actions, and only handling `TOGGLE` in the reducer.

Translator pattern

```
if (action.type === GOING) {
  dispatch({ type: SET_RSVP, payload: 'yes' });
}

if (action.type === NOT_GOING) {
  dispatch({ type: SET_RSVP, payload: 'no' });
}
```

Summary

In this chapter we presented a number of solutions to one of the most complicated issues in modern UI development: managing side effects. While Redux does not provide us with an opinionated solution out of the box, it is robust enough to allow different approaches to this problem. We've seen some well-established design patterns and examples of how we could use them to think about side effects and use middleware to organize them in a clear way.

In the next chapter we will cover WebSocket-based communication and how well it can work with the Redux architecture.

Chapter 10. WebSockets

WebSockets have brought a robust socket communication method directly into our browsers. What started as a solution for polling data changes on the server is slowly taking over more and more responsibilities from traditional REST endpoints. The action-based architecture of Redux makes working with WebSockets exceptionally easy and natural, as it involves using WebSockets as pipes to pass actions to and from the server.

Basic Architecture

WebSockets allow us to open a connection to a server and send or receive messages in a fully asynchronous way. The native implementation in browsers has only four callback methods that are required to fully support WebSockets:

- `onopen`—A connection has become active.
- `onclose`—A connection has been closed.
- `onerror`—An error related to WebSocket communication has been raised.
- `onmessage`—A new message has been received.

While multiple WebSockets might be used, most applications will require a single one or at most a few connections for different servers based on function (chat server, notifications server, etc.).

To start, we will build a system to communicate with a single WebSocket, which can later be extended for multi-WebSocket support.

Connecting to Redux

The general Redux architecture is all about sending well-defined messages to the store. This same scheme can work perfectly for server communication over WebSockets. The same structure of a plain object with the `type` property can be sent to the server, and we can receive a similarly structured response back:

Communication flow

```
> TO-SERVER: { type: 'GET_USER', id: 100 }
< FROM-SERVER: { type: 'USER_INFO', data: { ... }}
```

A more robust example might be a chat server, where we can dispatch to the store a message similar to `{ id: 'XXX', type: 'ADD_MESSAGE', msg: 'Hello' }`. Our store can handle this immediately by adding the message to the current messages array and send it "as is" over a WebSocket to the server. The server, in turn, can broadcast the message to all other clients. Each will get a perfectly standard Redux action that can be passed directly to its store.

This way our front end can use Redux actions to pass information between browser windows using the server as a generic dispatcher. Our server might do some additional work, like authentication and validation to prevent abuse, but in essence can act as a message passer.

An ideal WebSocket implementation for Redux would allow us to dispatch actions and have them smartly routed to the server when needed, and have any actions coming from the WebSocket be dispatched directly to the store.

Implementation

As with any infrastructure-related code, middleware is the perfect place for our WebSocket implementation. Using middleware will allow us to catch any actions that are required to be sent over the network and dispatch anything coming from the server. The basic WebSocket setup is as follows:

Basic structure of a WebSocket setup

```
const WS_ROOT = "ws://echo.websocket.org/";

const websocket = new WebSocket(WS_ROOT);

websocket.onopen    = () => {};
websocket.onclose   = () => {};
websocket.onerror   = event => {};
websocket.onmessage = event => {};
```

To make the code more readable, we can replace the four different assignments with a single use of `Object.assign()` and use code similar to this:

Using Object.assign()

```
Object.assign(websocket, {
  onopen()     { },
  onclose()    { },
  onerror(e)   { },
  onmessage(e) { }
});
```

In our middleware, we want to make sure a WebSocket is created only once. Thus, we cannot put the setup code inside the `action` handler:

The wrong way to initialize in middleware

```
const wsMiddleware = ({ dispatch, getState }) => next => action => {
  // Initialization not allowed
};
```

The code in the innermost block gets called every time an action is dispatched, so this would cause our setup and WebSocket creation code to be called multiple times. To prevent this, we can do the initialization outside the `action` callback block:

The correct way to initialize in middleware

```
const wsMiddleware = ({ dispatch, getState }) => next => {

  // TODO: Initialization

  return action => {
    // TODO: Middleware code
  };
};
```

Let's get back to the initialization code and consider how to handle each of the four callbacks: `onopen`, `onclose`, `onerror`, and `onmessage`.

onopen

This is mainly an informative stage. We need to indicate to our application that the socket is ready to send and receive data, and we might choose to notify the rest of the Redux application that the socket is ready (perhaps to show some indication in the UI).

Once the socket is open, we dispatch a simple `{ type: 'WS_CONNECTED' }` action:

Handling onopen

```
websocket.onopen = () => dispatch(wsConnected());
```

The `wsConnected()` function is a simple action creator that should be implemented in one of the action creator files:

app/actions/websocket.js

```
import { WS_CONNECTED } from 'constants/actionTypes';

const wsConnected = () => ({ type: WS_CONNECTED });
```

onclose

The close or disconnect event is very similar to `onopen` and can be handled in the exact same way:

app/actions/websocket.js

```
import * as actions from 'constants/actionTypes';

const wsCconnected = () => ({ type: actions.WS_CONNECTED });
const wsDisconnected = () => ({ type: actions.WS_DISCONNECTED });
```

Handling onclose

```
websocket.onclose = () => dispatch(wsDisconnected());
```

onerror

The WebSocket implementation in a browser can provide information on various failures in the underlying socket communication. Handling these errors is similar to handling regular REST API errors and might involve dispatching an action to update the UI or closing the socket if needed.

In this example we will stop at a generic `console.log()` and leave it to the reader to consider more advanced error handling methods:

Handling onerror

```
Websocket.onerror = (error) =>
  console.log("WS Error", error.data);
```

onmessage

This callback is called every time a new message is received over a WebSocket. If we have built our server to be fully compatible with Redux actions, the message can simply be dispatched to the store:

Handling onmessage

```
websocket.onmessage = (event) => dispatch(JSON.parse(event.data));
```

Handling Outgoing Messages and Actions

With all the WebSocket callbacks handled, we need to consider how and when to pass actions from Redux to the server:

A function to hold the WebSocket functionality

```
return action => {
  // TODO: Pass action to server

  next(action);
};
```

Before sending any actions, we need to make sure that the WebSocket is open and ready for transmissions. WebSockets have a `readyState` property that returns the current socket status:

Checking if the socket is open

```
const SOCKET_STATES = {
  CONNECTING: 0,
  OPEN: 1,
  CLOSING: 2,
  CLOSED: 3
};

if (websocket.readyState === SOCKET_STATES.OPEN) {
  // Send
}
```

Even when the socket is open, not all actions have to be sent (for example, actions like TAB_SELECTED or REST_API_COMPLETE). It is best to leave the decision of what to send to our action creators. The standard way to provide special information about actions to middleware is to use the meta key inside an action. Thus, instead of using a regular action creator:

A regular action creator

```
export const localAction = (data) => ({
  type: TEST,
  data
});
```

we can add special information to the metadata part of the action:

An action creator to use in websocket

```
export const serverAction = (data) => ({
  type: TEST,
  data,
  meta: { websocket: true }
});
```

This way our middleware can use the meta.websocket field to decide whether to pass the action on or not:

Sending actions to the server

```
return action => {
  if (websocket.readyState === SOCKET_STATES.OPEN &&
      action.meta &&
      action.meta.websocket) {
    websocket.send(JSON.stringify(action));
  }

  next(action);
};
```

Note, however, that this code might cause a surprising bug. Since we are sending the whole action to the server, it might in turn broadcast it to all other clients (even ourselves). And because we didn't remove the action's `meta` information, the other clients' WebSocket middleware might rebroadcast it again and again.

A Redux-aware server should consider stripping all `meta` information for any action it receives. In our implementation we will remove this on the client side, though the server should still do the check:

Sending actions to server without metadata

```
return next => action => {
  if (websocket.readyState === SOCKET_STATES.OPEN &&
      action.meta &&
      action.meta.websocket) {

    // Remove action metadata before sending
    const cleanAction = Object.assign({}, action, { meta: undefined });
    websocket.send(JSON.stringify(cleanAction));
  }

  next(action);
};
```

Using this approach, sending actions to our server via a WebSocket becomes as simple as setting the `meta.websocket` field to `true`.

Complete WebSocket Middleware Code

middleware/ws.js

```
import { wsConnected, wsDisconnected } from 'actions';
import { WS_ROOT } from 'const/global';

const SOCKET_STATES = {
  CONNECTING: 0,
  OPEN: 1,
  CLOSING: 2,
  CLOSED: 3
};

const wsMiddleware = ({ dispatch }) => next => {

 const websocket = new WebSocket(WS_ROOT);

 Object.assign(websocket, {
   onopen: () => dispatch(wsConnected()),
   onclose: () => dispatch(wsDisconnected()),
   onerror: (error) => console.log(`WS Error: ${ error.data }`),
   onmessage: (event) => dispatch(JSON.parse(event.data))
 });

 return action => {
   if (websocket.readyState === SOCKET_STATES.OPEN &&
       action.meta &&
       action.meta.websocket) {

     // Remove action metadata before sending
     const cleanAction = Object.assign({}, action, {
       meta: undefined
     });

     websocket.send(JSON.stringify(cleanAction));
   }

   next(action);
};

export default wsMiddleware;
```

Authentication

Handling authentication with WebSockets can be a little tricky, as in many applications WebSockets are used alongside regular HTTP requests. The authentication will usually be done via regular REST or OAUTH calls and the front end granted a token either set in cookies or to be saved in `localStorage`.

To allow the server to authenticate a WebSocket, a special agreed-upon action has to be sent by the client. In the case of Redux, a special action object can be serialized and sent before doing any other work over WebSockets.

Sample Flow

A simple way to implement authentication might be to send an API action to our server containing an email address and a password:

Action to authenticate with the server

```
dispatch({
  type: API,
  payload: {
    url: 'login',
    method: 'POST',
    success: loggedIn,
    data: {
      email: 'info@redux-book.com',
      password: 'top secret'
    }
  }
});
```

If successful, our API middleware will dispatch the `LOGIN_SUCCESS` action containing the information returned from the server:

Action dispatched on successful login

```
{
  type: LOGGED_IN,
  payload: { token: 'xxxYYYzzzz' }
}
```

Our user's reducer will probably act on this action to add the token to the state, to be passed in the headers of future API requests to the server.

To make WebSockets authenticate using this token, we can add special code to our WebSocket API that will act upon `LOGGED_IN` and `LOGOUT` actions:

Authentication code in the WebSocket middleware

```
if (action.type === LOGGED_IN) {
  dispatch({
    type: WEBSOCKET_AUTH,
    payload: action.payload.token,
    meta: { websocket: true }
  });
}

if (action.type === LOGOUT) {
  dispatch({
    type: WEBSOCKET_LOGOUT,
    meta: { websocket: true }
  });
}
```

Now the passage of `LOGGED_IN` will cause a new WebSocket-enabled action to be dispatched and processed by our middleware to authenticate with the server:

The flow of actions

```
> Store:
{ type: API, payload: ... }

> Server:
POST http://.../login

> Store:
{ type: LOGGED_IN, payload: token }

> Store:
{ type: WEBSOCKET_AUTH, payload: token, meta: { websocket: true }}

> WebSocket:
{ type: WEBSOCKET_AUTH, payload: token }
```

Notes

In a more sophisticated implementation of the WebSocket middleware, it would be best to keep track of the authentication state of the WebSocket and prevent actions from being sent or received before the WebSocket has been authenticated or after it has been logged out from.

When the token is already present in a cookie, it will be passed to the WebSocket as soon as the socket is opened. This might cause problems if the login process happens after the application loads—or, even worse, when the user logs out the WebSocket might remain authenticated. It is better to use the action-based authentication approach described here to avoid these and similar issues.

Summary

This chapter has illustrated how well WebSockets work with Redux and the practical steps needed to set up WebSocket-based communication.

In the next chapter we will cover the subject of testing and how each part of our Redux application can be tested, both separately and together.

Chapter 11. Testing

One of the key strengths of Redux is its testability. We can create automated unit tests for each of the different actors (reducers, action creators, and middleware), and combine them together for comprehensive integration testing.

There are a large number of testing tools available. We will be using the Jest[67] library from Facebook here, the latest version of which proves to be an excellent choice for testing Redux; however, the exact tooling you use is relatively unimportant as most parts of our Redux application will rely on plain JavaScript functions and objects with no complicated libraries or async flows to test. Use of other frameworks and tools such as Karma, Mocha, and so on should look very similar to the examples in this chapter.

To find out how to install and work with Jest, check out the getting started guide[68].

Directory Organization

To start off, we need a way to organize our tests. There are two main approaches: putting the tests together in the same directory with the implementation files, or putting them in a separate directory. We will use the latter approach, but the choice is largely a matter of convenience and personal preference, with the only side effects being different test runner configurations.

We will create a separate test file for each implementation file in our project. In the case of *app/actions/recipes.js*, our test file will be *tests/actions/recipes.test.js*.

[67]https://facebook.github.io/jest/

[68]https://facebook.github.io/jest/docs/en/getting-started.html

File Structure

In our test files we will use a `describe()` function to wrap all our tests. The first string parameter to this function will allow us to easily determine which groups of tests are failing or succeeding:

Sample test file structure

```
describe('actions/recipes', () => {
  // TODO: Add tests
});
```

Inside this function other nested `describe()` functions can be used, to further distinguish between different sets of states (for example, testing failing or succeeding API calls). Each test in Jest is wrapped in an `it()` block describing what the test does. To keep the tests readable and easy to understand, it is generally recommended to create as many short tests as possible (each within its own `it()` block) rather than creating very large single test functions:

Sample test file with test placeholders

```
describe('actions/recipes', () => {
  it('addRecipe', () => {/* TODO: Implement test */});

  it('deleteRecipe', () => {/* TODO: Implement test */});
});
```

Testing Action Creators

Throughout this book we have tried to keep asynchronous flows out of action creators by moving them into middleware and utility functions. This approach allows for very easy testing of action creators, as they are functions that return plain JavaScript objects:

Simple action creator

```
import * as actions from 'constants/actionTypes';

export const setRecipes = recipes => ({
 type: actions.SET_RECIPES,
 payload: recipes
});
```

Our `setRecipes()` action creator receives a single parameter and returns a plain JavaScript object. Since there is no control flow logic or side effects, any call to this function will always return the same value, making it very easy to test:

Simple test for addRecipe

```
import * as actions from 'actions'

describe('actions/recipes', () => {
 it('addRecipe', () => {
   const expected = { type: 'ADD_RECIPE', payload: 'test' };
   const actual = actions.addRecipe('test');

   expect(actual).toEqual(expected);
 });
});
```

This test is built in three parts. First, we calculate what our action creator should return when called with `'test'` as an argument—in this case a JavaScript object containing two keys, `type` and `payload`:

Calculating the expected result

```
const expected = { type: 'ADD_RECIPE', payload: 'test' };
```

The second stage is running the action creator `actions.addRecipe('test')` to get the value built by our action creator's implementation:

Calculating the actual result

```
const actual = actions.addRecipe('test');
```

And the final stage is using Jest's `expect()` and `toEqual()` functions to verify that the actual and expected results are the same:

Verifying matching results

```
expect(actual).toEqual(expected);
```

If the `expected` and `actual` objects differ, Jest will throw an error and provide information describing the differences, allowing us to catch incorrect implementations.

Improving the Code

Due to the simplicity of this code, it is common to combine multiple stages into a single call and rewrite the test as follows:

Shorter version of the test

```
it('ADD_RECIPE', () => {
  const expected = { type: 'ADD_RECIPE', payload: 'test' };

  expect(actions.addRecipe('test')).toEqual(expected);
});
```

Using Snapshots

The approach of calculating the expected value and then comparing it to dynamically calculated values is very common in Redux tests. To save typing time and make the code cleaner to read, we can use one of Jest's greatest features, snapshots[69].

Instead of building the expected result, we can ask Jest to run the `expect()` block and save the result in a special *.snap* file, generating our `expected` object automatically and managing it for us:

Test with snapshot

```
it('ADD_RECIPE', () => {
  expect(actions.addRecipe('test')).toMatchSnapshot();
});
```

The `expected` calculation is gone, and instead of using `isEqual()`, Jest will now compare the result of the expression inside `expect()` to a version it has saved on disk. The actual snapshot is placed in a *__snapshots__* directory in a file with the same name as the test file plus the *.snap* extension:

snapshots/action.test.js.snap

```
exports[`actions ADD_RECIPE 1`] = `
Object {
  "payload": "test",
  "type": "ADD_RECIPE",
}
`;
```

[69] https://facebook.github.io/jest/docs/en/tutorial-react.html#snapshot-testing

The structure is more complicated than that of a regular JavaScript object, but the result is exactly the same as our original `expected` calculation:

Calculating the expected result

```
const expected = { type: 'ADD_RECIPE', payload: 'test' };
```

What happens when our code changes? In some cases we want to intentionally change the structure of our action object. In these cases, Jest will detect that the returned value does not match what is saved inside its snapshot file and throw an error. But if we determine that the new result is the correct one and the cached snapshot is no longer valid, we can easily tell Jest to update its snapshot version to the new one.

Dynamic Action Creators

In some cases, action creators might contain logic that emits different actions based on the input parameters. As long as no asynchronous code or other external entities like `localStorage` are touched, we can easily test the logic by providing different input parameters to the action creator and verifying that it creates the correct object every time:

An action creator that modifies the input

```
export const addRecipe = (title) => ({
  type: actions.ADD_RECIPE,
  payload: title || "Default"
});
```

The modified `addRecipe()` action creator will set `payload` to `"Default"` if the user does not provide a title. To test this behavior we can create two tests, one that provides a parameter (as we already did) and one that provides an empty string. A fully comprehensive test might contain multiple “empty string” cases, for `null`, `undefined`, and `''`:

Combined test of multiple falsy input values

```
it('ADD_RECIPE', () => {
  expect(actions.addRecipe(undefined)).toMatchSnapshot();
  expect(actions.addRecipe(null)).toMatchSnapshot();
  expect(actions.addRecipe('')).toMatchSnapshot();
});
```

In contrast to what we discussed earlier, here we tried putting multiple `expect()` functions into the same test. While this approach will work, it will be harder to identify which of the test cases failed in the event of an error.

Since we are using JavaScript to write our tests, we can easily create test cases for each input value without increasing our code size significantly (by creating an `it()` clause for each). We can do that by adding all the possible inputs into an array and automatically creating corresponding `it()` blocks:

Automatically creating tests for each test case

```
[undefined, null, ''].forEach((param) =>
  it(`ADD_RECIPE with ${param}` , () => {
    expect(actions.addRecipe(param)).toMatchSnapshot()
  }));
```

Using this approach we get three different `it()` blocks automatically generated by JavaScript, keeping our tests clear and the code short.

Async Action Creators

Throughout this book we have tried to discourage the use of asynchronous action creators and functions that have side effects and have used various libraries, like `redux-thunk`[70], to allow action creators to schedule async work and be non-idempotent. One of the main benefits is the ease of testing regular action creators offer. More information on this can be found in the Actions and Action Creators chapter.

For our example, we will create a simple async action creator that uses `redux-thunk` and the `axios` API to get recipe data from a server and dispatches the result with a `SET_RECIPE` action:

Async action creator

```
export const setRecipe = (id, data) => ({
  type: actions.SET_RECIPE,
  payload: { id, data }
});

export const fetchRecipe = id => dispatch => {
  return axios.get('recipe/' + id)
    .then(({ data }) => dispatch(setRecipe(id, data)))
};
```

[70] https://github.com/gaearon/redux-thunk

With `redux-thunk`, our action creators can return a function instead of a plain JavaScript object. The thunk middleware will call such a function with the `dispatch()` and `getState()` methods as arguments. This allows the action creator to use the `axios` library to get data from the server and dispatch an action when it's done using `dispatch()`.

Mocking axios

Before we move on to the actual tests, we have to lay out some infrastructure to mock the `axios` library. With Jest, we can create overrides for any imported library using the built-in mocking features.

To do this we need to create a *__mocks__* directory at the same level in the project as the *node_modules* directory. Inside we can place files named to override any Node package. For example, the *axios.js* file will override `import axios from 'axios'` in our project.

Next, we need to update the *package.json* file to include a `jest` key with an `axios` property pointing to the `axios` mock directory:

package.json with overrides

```
"jest": {
    "axios": "<rootDir>/__mocks__/axios.js"
  }
}
```

Now, our mocked `axios` library will allow us to mock successful and failing API requests:

axios mock

```
let mocked = null;

const noMocks = () => throw('No request mocked');

export const mockResponse = (status, statusText, data) => ({
  data,
  status,
  statusText
});

const handleResponse = (mockedUrl, response) =>
  mocked = jest.fn().mockImplementation(({ url }) => {
    mocked = null;

    if (url === mockedUrl) {
      return response;
    }

    throw('Unknown URL: ' + url);
  });

export const clearMock = () => mocked = null;

export const mockRequest = (mockedUrl, status, data) =>
  handleResponse(
    mockedUrl,
    Promise.resolve(mockResponse(status, null, data)));

export const mockRequestError = (mockedUrl, state, error) =>
  handleResponse(
    mockedUrl,
    Promise.reject(mockResponse(state, error, '{}')));

export default {
  request: (params) => (mocked || noMocks)(params),
  get: (url) => (mocked || noMocks)({ url }),
  name: 'mock'
};
```

The main two APIs we use are the `mockRequest()` and `mockRequestError()` functions. After a call to the mock, a promise is created to be returned once `axios` is used by our code:

Sample mocked server access code

```
import { mockRequest } from 'axios';
import axios from 'axios';

mockRequest('http://redux-book.com/status.json', 200, { status: 'ok' });

axios.get('http://redux-book.com/status.json')
  .then(data => console.log(data));
```

In this example we mock access to `'http://redux-book.com/status.json'`, making an `axios.get()` request to return `{ status: 'ok' }` as the data.

Creating a Mock Store

Unlike with simple action creators, our code now relies on `dispatch()` being used, which forces us to create a mock instance of a store. To do so, we will use the `redux-mock-store`[71] library:

Creating a mock store

```
import configureStore from 'redux-mock-store';
import thunk from 'redux-thunk';

const mockStore = configureStore([ thunk ]);
```

Here we create a mock store object with a single thunk middleware. This store object can be used as a regular Redux store; it supports dispatching of actions and will later allow us to assert that the correct set of actions was sent to our store.

[71] https://github.com/arnaudbenard/redux-mock-store

Async Action Creator Test Structure

Since async action creators might contain different flows based on the result of the async action, it is best to put them into their own `describe()` blocks in the tests. This will also allow us to easily create a fresh "mock store" for each of the test cases using Jest's `beforeEach()` method:

Structure of an async test block

```
describe('actions/recipes', () => {
  let store;

  beforeEach(() => store = mockStore({}));

  it('fetchRecipe');

  ...
});
```

Our mock store gets automatically recreated before each iteration of the tests, clearing any actions cached from the previous run.

Basic Async Action Creator Test

Jest handles async tests by allowing us to return a promise as the test's result. If a promise is returned, the test runner will wait for the promise to resolve and only then continue to the next test:

Async test

```
it('FETCH_RECIPE', () => {
  return store.dispatch(actions.fetchRecipe(100));
});
```

Since `store.dispatch()` in this case returns a promise (remember, our `fetchRecipe()` action creator returns a call to the `axios` library), we can use it to create an async test.

To add an `expect()` clause to the code, we can use the same promise and run our tests as soon as it is resolved:

Adding expect() clause to async tests

```
it('FETCH_RECIPE', () => {
  return store.dispatch(actions.fetchRecipe(100))
    .then(() => expect(store.getActions()).toEqual([]))
});
```

The `expect()` clause is similar to what we used in our previous tests. We are using the mocked store's `getActions()` method to get an array of all the actions dispatched to the store. In our implementation we expect a successful network call to dispatch the result of the `setRecipe()` action creator.

Running this test now will fail, since we didn't instruct our mocked `axios` library to mock the target URL. Using the small utility library we created previously, we can create the mock that will result in the correct action sequence:

Full async test

```
it('FETCH_RECIPE', () => {
  mockRequest('recipe/100', 200, '{"title":"hello"}');

  return store.dispatch(actions.fetchRecipe(100))
    .then(() => expect(store.getActions()).toMatchSnapshot())
});
```

Here we mock a `200` successful response from the `axios` library and expect that dispatching the async action created by `fetchRecipe(100)` results in a later dispatch of the action created by `setRecipe()`.

Async Tests Summary

As can be seen from this minimal example, testing async action creators is much more complicated than testing regular action creators. If we add proper error handling and branching, the tests quickly become very hard to build and reason about.

The Middleware chapter offers an alternative to async action creators by moving the asynchronous logic into middleware. This allows us to test and concentrate all the async code of an application in a single place.

Testing Reducers

Testing reducers is very similar to testing action creators, as reducers by definition are idempotent (given a state and an action, the same new state will be returned every time). This makes reducer tests very easy to write, as we simply need to call the reducer with different combinations of input to verify the correctness of the output.

Basic Reducer Test

For our test, we can create a very crude reducer that handles a single `ADD_RECIPE` action and whose state is simply an array of recipes:

Simple recipes reducer

```
import { ADD_RECIPE } from 'constants.js';

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_RECIPE:
      return state.concat({ title: action.payload });
  }

  return state;
};

export default reducer;
```

There are two main test cases to consider, adding a recipe to an empty list and a non-empty one. We can test the first case as follows:

Simple recipes reducer test

```
import reducer from 'reducers/recipes';

import { ADD_RECIPE } from 'constants';

describe('reducers/recipes', () => {
  it('should add recipe to empty list', () => {
    const initialState = [];
    const action       = { type: ADD_RECIPE, payload: 'test' };
    const expected     = [{ title: "test" }];
    const actual       = reducer(initialState, action);

    expect(actual).toEqual(expected);
  });
};
```

The steps taken here should already be familiar:

1. Calculate the initial state (an empty array in our case).
2. Build the action to send.
3. Set the `expected` state the reducer should return.
4. Call `reducer()` to calculate the state based on the empty array and our action.
5. Verify that the `actual` and `expected` states match.
6. Calculate the initial state.

Before we simplify the code, let's consider the second test case, adding a recipe to a non-empty list:

Test of adding a recipe to a non-empty list

```
it('should add recipe to non-empty list', () => {
  const initialState = [{ title: "first" }];
  const action       = { type: ADD_RECIPE, payload: 'test' };
  const expected     = [{ title: "first" }, { title: "test" }];
  const actual       = reducer(initialState, action);

  expect(actual).toEqual(expected);
});
```

In this test we start with a list containing a single item and update our `expected` result to match. While this works, it has a maintenance problem. What will happen if our recipes contain more fields in the future?

Using this method of writing tests, we will need to find each test definition's initial state and add more properties to it. This complicates the test writer's job without providing any benefits. Luckily, we already have a way to create non-empty states: the reducer! Since we already tested adding to an empty list in the first test, we can rely on our reducer to create a non-empty list with all the required recipe information:

Building initial state using the reducer

```
const initialState = reducer([], { type: ADD_RECIPE, payload: 'first' });
```

This only partially solves the problem, though, as we are still treating the initial state as an empty array (`[]`). While this is true in our test case, other reducers might have more complicated structures to deal with. A simple solution would be to create a `const initialState = {}` at the root of the tests and rely on it when needed:

Setting initial state for all tests

```
describe('reducers/recipes', () => {
  const initialState = [];

  it('should add recipe to empty list', () => {
    const action   = { type: ADD_RECIPE, payload: 'test' };
    const expected = [{ title: "test" }];
    const actual   = reducer(initialState, action);

    expect(actual).toEqual(expected);
  });

  it('should add recipe to non-empty list', () => {
    const testState = reducer(initialState,
        { type: ADD_RECIPE, payload: 'first' });
    const action   = { type: ADD_RECIPE, payload: 'test' };
    const expected = [{ title: "first" }, { title: "test" }];
    const actual   = reducer(testState, action);

    expect(actual).toEqual(expected);
  });
};
```

The same `initialState` is used in all the tests, but it is still hardcoded in our test file. If our reducer changes the way state is built, we will be forced to update the test files accordingly. To remove this dependency we can rely on a feature that is enforced by Redux's `combineReducers()`. It mandates that any reducer called with an `undefined` state must return its part of the initial state structure:

Excerpt from our reducer

```
const initialState = [];

const reducer = (state = initialState, action) => {
  ...
};
```

This means we can use the reducer to get the initial state to use for all of our tests by calling it with `undefined` and any action:

Generating the initial state using our reducer

```
const initialState = reducer(undefined, { type: 'INIT' });
```

The result will put the same `[]` in the initial state, but now any changes to what the reducer considers to be the initial state will be automatically picked up by the tests as well.

Making the Tests Pretty

Now that we've solved all the functionality issues, we can use the same tricks we used in the action creator tests to simplify our reducer tests. Here are the original tests:

Original tests

```
it('should add recipe to empty list', () => {
  const action = { type: ADD_RECIPE, payload: 'test' };
  const expected = [{ title: "test" }];

  expect(reducer(initialState, action)).toEqual(expected);
});

it('should add recipe to non-empty list', () => {
  const baseState = reducer(initialState,
    { type: ADD_RECIPE, payload: 'first' });
  const action   = { type: ADD_RECIPE, payload: 'test' };
  const expected = [{ title: "first" }, { title: "test" }];
  const actual   = reducer(baseState, action);

  expect(actual).toEqual(expected);
});
```

The first step will be to combine `action`, `actual`, and `expect()` into a single line:

Simplified tests

```
it('should add recipe to empty list', () => {
  const expected = [{ title: "test" }];

  expect(reducer(initialState, { type: ADD_RECIPE, payload: 'test' }))
    .toEqual(expected);
});

it('should add recipe to non-empty list', () => {
  const baseState = reducer(initialState, { type: ADD_RECIPE, payload: 'first' }\
);
  const expected  = [{ title: "first" }, { title: "test" }];

  expect(reducer(baseState, { type: ADD_RECIPE, payload: 'test' }))
    .toEqual(expected);
});
```

The second step is to use Jest's snapshots instead of manually calculated `expected` values:

Simplified tests, stage 2

```
it('should add recipe to empty list', () => {
  expect(reducer(initialState,
    { type: ADD_RECIPE, payload: 'test' }))
    .toMatchSnapshot()
});

it('should add recipe to non-empty list', () => {
  const baseState = reducer(initialState,
    { type: ADD_RECIPE, payload: 'first' });

  expect(reducer(baseState, { type: ADD_RECIPE, payload: 'test' }))
    .toMatchSnapshot();
});
```

Avoiding Mutations

One key requirement is that our reducers never modify the state, but only create a new one. Our current tests do not verify this behavior (try changing `.concat()` to `.push()` in the reducer implementation).

While we can try to catch these mistakes by manually verifying that the initial state did not change, a simpler approach would be to "freeze" the initial state and have any changes to it automatically stop the tests. To achieve this we can use the excellent `deep-freeze`[72] library, installed as follows:

Installing deep-freeze

```
npm install deep-freeze --save
```

To use `deep-freeze`, we wrap our initial state with a call to `deepFreeze()`:

Using deep-freeze

```
import deepFreeze from 'deep-freeze';

const initialState = deepFreeze(reducer(undefined, { type: 'INIT' }));
```

[72] https://github.com/substack/deep-freeze

Any attempt by any parts of our code to modify `initialState` will now automatically throw an error:

Automatically catching change attempts

```
initialState.push('test');
> TypeError: Can't add property 0, object is not extensible
```

To ensure that our reducers never change the original state, we can always call `deepFreeze()` on the state before passing it on to the reducer:

Updated adding to non-empty list test

```
it('should add recipe to non-empty list', () => {
  const initAction = { type: ADD_RECIPE, payload: 'first' };
  const baseState  = deepFreeze(reducer(initialState, initAction));

  expect(reducer(baseState, { type: ADD_RECIPE, payload: 'test' }))
    .toMatchSnapshot();
});
```

Action Creators and Reducers

Usually when writing unit tests it is recommended to test each part of the system separately and only test connections and interfaces in the integration tests. In the case of Redux, though, it is worth considering the connection between action creators and reducers.

In the current way we've built our tests, the `ADD_RECIPE` action object is defined in three different places: the recipe's action creators, the recipe's tests, and the reducer's tests.

If, in the future, we decide to change the structure of the `ADD_RECIPE` action, our action creator tests will catch the change and remind us to update the test code. But the reducer's tests will continue to pass unless we remember to change the hardcoded `ADD_RECIPE` action objects used in those tests as well.

This can lead to painful edge cases where all the tests pass, but the system doesn't work. To avoid this, we can stop using hardcoded action objects in reducers and rely on action creators directly:

Reducer tests modified to use action creators directly

```
it('should add recipe to empty list', () => {
  expect(reducer(initialState, addRecipe('test'))).toMatchSnapshot()
});

it('should add recipe to non-empty list', () => {
  const baseState = deepFreeze(reducer(initialState, addRecipe('first')));

  expect(reducer(baseState, addRecipe('test'))).toMatchSnapshot();
});
```

While somewhat breaking the unit test principle, combining the reducers with action creators results in cleaner code, fewer bugs, and less duplication.

Unknown Actions

One last issue to test with reducers is that they gracefully handle unknown actions and return the original state passed to them without modifications.

Since every action can propagate to the whole reducer tree, it is important for the reducer to return the original state and not a modified copy. This will allow UI libraries to identify changes in the tree using reference comparison.

We can do this as follows:

Unknown actions test

```
it('should handle unknown actions', () => {
  expect(reducer(initialState, { type: 'FAKE' })).toBe(initialState);
});
```

An important thing to note about this test is the use of `.toBe()` instead of `.toEqual()` or `.toMatchSnapshot()`. Unlike the other methods, `.toBe()` expects the result of the reducer to be the exact same object, not a similar object with the same data:

Example use of toBe()

```
const a = { name: 'Kipi' };
const b = { name: 'Kipi' };

it('passing test', () => expect(a).toEqual(b));

it('failing test', () => expect(a).toBe(b));
```

The main goal of this test is to verify that our reducer returns the original state if the action sent was not intended for it:

Correct reducer code

```
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case ADD_RECIPE: return state.concat({ title: action.payload })
  }

  return state;
};
```

Testing Middleware

The middleware are where most of the complex logic of our application will reside. Since they have full access to the store's `dispatch()` and `getState()` methods as well as control over the actions' flow via `next()`, middleware can become quite complex, with nontrivial asynchronous flows.

Middleware Test Structure

At their core middleware are functions that receive actions to process, albeit with a complicated signature (a function returning a function returning a function). The first two function calls are made by Redux during startup. Only the last call is made dynamically, when new actions need to be processed.

The basic signature looks like this:

Middleware signature

```
function sampleMiddleware({ dispatch, getState }) {
  return function nextWrapper(next) {
    return function innerCode(action) {
      // TODO: Implement the middleware

      next(action);
    }
  }
}
```

To test middleware we will need to mock `dispatch()`, `getState()`, and `mock()` and call `sampleMiddleware()` and `nextWrapper()` to get our test target, the `innerCode()` function:

Setting up middleware for tests

```
const next       = jest.fn();
const dispatch   = jest.fn();
const getState   = jest.fn();
const middleware = apiMiddleware({ dispatch, getState })(next);
```

We can now use the regular Jest tests to test the `middleware()` function we built by calling it with action objects:

Simple middleware test call

```
it('should process action', () => {
  const next       = jest.fn();
  const dispatch   = jest.fn();
  const getState   = jest.fn();
  const middleware = sampleMiddleware({ dispatch, getState })(next);

  const sampleAction = { type: 'SAMPLE_ACTION' };

  middleware(sampleAction);

  // TODO: Add expects
});
```

In the case of our simple middleware, we only want to verify that it passed the action correctly down the chain by calling `next(action)`. Since we used Jest's function mocking, we can get a full history of calls to each mock by accessing `next.mock.calls`:

Verifying correct calls to next()

```
expect(next.mock.calls.length).toBe(1);
expect(next.mock.calls[0].length).toBe(1);
expect(next.mock.calls[0][0]).toEqual(sampleAction);
```

Our test verified that there was only one call to `next()`. In that call there was only one parameter passed, and that parameter was the sample action.

We could do all the three tests in one go by using:

Combining the test cases

```
expect(next.mock.calls).toEqual([[sampleAction]]);
```

Simplifying the Test Structure

To avoid duplicating the test setup before every `it()` clause, we can use Jest's `beforeEach()` method to combine the setup in one place:

Generic setup

```
describe('sample middleware', () => {
  let next, dispatch, getState, middleware;

  beforeEach(() => {
    next       = jest.fn();
    dispatch   = jest.fn();
    getState   = jest.fn();
    middleware = sampleMiddleware({ dispatch, getState })(next);
  });

  it('should process action', () => {
    const sampleAction = { type: 'SAMPLE_ACTION' };

    middleware(sampleAction);

    expect(next.mock.calls).toEqual([[sampleAction]]);
  };
};
```

Using this structure, our middleware will be rebuilt before each test and all the mocked functions will be reset, keeping the testing code itself as short as possible.

Testing Async Middleware

For a more complete example, let's use an API middleware similar to the one discussed in the Server Communication chapter:

API middleware

```
import axios from 'axios';
import { API } from 'consts';
import { apiStarted, apiFinished, apiError } from 'actions/ui';

const apiMiddleware = ({ dispatch, getState }) => next => action => {
  if (action.type !== API) {
    return next(action);
  }

  const { url, success } = action.payload;
  const headers = {};
  const accessToken = (getState() || {}).accessToken;

  if (accessToken) {
    headers['Access-Token'] = accessToken;
  }

  dispatch(apiStarted());

  return axios.request({ url, headers })
    .then(response => {
      dispatch(success(JSON.parse(response.data)));
      dispatch(apiFinished());
    })
    .catch(params => dispatch(apiError(new Error(params))));
};

export default apiMiddleware;
```

Our middleware catches any actions of type `'API'`, which must contain a `payload` key with a `url` to make a request to and a `success` parameter that holds an action creator to call with the returned data:

Sample API action

```
const setData = data => ({
  type: 'SET_DATA',
  payload: data
});

const apiAction = () => ({
  type: API,
  payload: {
    success: setData,
    url: 'fake.json'
  }
});
```

In our Redux call, calling `dispatch(apiAction())` will result in our API middleware doing a `GET` request for *server/fake.json* and, if successful, dispatching the `SET_DATA` action with the response set as `payload`. When there is an error, an action created by `apiError()` will be dispatched containing `status` and `statusText`.

Another important feature of the API middleware is that it will dispatch `apiStarted()` before contacting the server and `apiFinished()` on success (or `apiError()` on failure). This allows the application to keep track of the number of active requests to the server and display a spinner or some other user indication.

To fully test this middleware we can split the tests into three groups: general tests, success tests, and failure tests.

Setup

To make our tests cleaner we will be using the structure discussed previously and mocking the `axios` API as discussed in the "Async Action Creators" section of this chapter.

We will also use the sample API action creators from earlier to drive the tests and a fake data response from the server:

Base for the API middleware tests

```
import apiMiddleware from 'middleware/api';
import { mockRequest, mockRequestError } from 'axios';
import { API_STARTED, API_FINISHED, API, API_ERROR } from 'constants';

const data = { title: 'hello' };

const setData = data => ({
  type: 'SET_DATA',
  payload: data
});

const apiAction = () => ({
  type: API,
  payload: {
    success: setData,
    url: 'fake.json'
  }
});

describe("api middleware", () => {
  let next, dispatch, middleware;

  beforeEach(() => {
    next       = jest.fn();
    dispatch   = jest.fn();
    middleware = apiMiddleware({ dispatch })(next);
  });

  describe('general', () => { /* TODO */ });
  describe('success', () => { /* TODO */ });
  describe('error', () => { /* TODO */ });
});
```

General tests

The first test for any middleware is to ensure that it passes unknown actions down the chain. If we forget to use `next(action)`, no actions will reach the reducers:

Verifying unknown actions are handled correctly

```
it('should ignore non-API actions', () => {
  const sampleAction = { type: 'SAMPLE_ACTION' };

  middleware(sampleAction);

  expect(dispatch.mock.calls.length).toBe(0);
  expect(next.mock.calls).toEqual([[sampleAction]]);
});
```

Here we verify that `dispatch()` is never called and `next()` is called exactly once with our `sampleAction`. Since we will be using `dispatch.mock.calls` and `next.mock.calls` very often in our tests, we can shorten them a little by adding the following to our setup code:

Improving the setup code

```
let next, dispatch, middleware, dispatchCalls, nextCalls;

beforeEach(() => {
  next          = jest.fn();
  dispatch      = jest.fn();

  dispatchCalls = dispatch.mock.calls;
  nextCalls     = next.mock.calls;

  middleware    = apiMiddleware({ dispatch })(next);
});
```

Now instead of `expect(next.mock.calls)` we can use `expect(nextCalls)`.

Another general test could be to verify that the API_STARTED action is dispatched every time the middleware is about to access the server:

Testing that API_STARTED is dispatched

```
it('should dispatch API_STARTED', () => {
  middleware(apiAction());
  expect(dispatchCalls[0]).toEqual([{ type: API_STARTED }]);
});
```

Our expect() call only checks that the first dispatch() action is API_STARTED because the middleware might call additional actions later on.

Successful server access

In the success scenario, we need to mock the axios library to return a successful response. We will be using the same mockRequest() utility introduced in the "Async Action Creators" section of this chapter.

Our basic success tests need to check that API_FINISHED is dispatched once the API is done and that our success() action creator is called, passed the response, and dispatched to the store:

Success tests framework

```
describe('success', () => {
  beforeEach(() => mockRequest('recipes.json', 200, JSON.stringify(data)));

  it('should dispatch API_FINISHED');

  it('should dispatch SET_DATA');
});
```

A first attempt at testing the first case might look similar to the API_STARTED test:

Testing that API_FINISHED is dispatched

```
it('should dispatch API_FINISHED', () => {
  middleware(apiAction());
  expect(dispatchCalls[2]).toEqual([{ type: API_FINISHED }]);
});
```

Unfortunately, this code will not work. Since API_FINISHED is only dispatched after the API's promise is resolved, we need to wait for that to happen before calling expect().

As discussed in "Async Action Creators," we rely on our call to the middleware to return a promise that gets resolved once the network call completes. Only then can we run assertions and verify that everything behaved according to our expectations:

The correct API_FINISHED test

```
it('should dispatch API_FINISHED', () => {
  return middleware(apiAction())
    .then(() => {
      expect(dispatchCalls[2]).toEqual([{ type: API_FINISHED }]);
    });
});
```

In this version of the test, only once the promise returned by the call to `middleware()` is resolved do we check the array of calls to `dispatch()`. Since our new test is a one-liner, we can use some ES2015 magic and Jest's `toMatchSnapshot()` method to shorten the code:

Shorter API_FINISHED test

```
it('should dispatch API_FINISHED', () =>
   middleware(apiAction()).then(() =>
     expect(dispatchCalls[2]).toMatchSnapshot()));
```

Testing that the API middleware correctly sends the response from the server via the action creator provided in `action.payload.success` is very similar:

Testing that SET_DATA was dispatched

```
it('should dispatch SET_DATA', () =>
  middleware(apiAction()).then(() =>
     expect(dispatchCalls[1]).toEqual([setData(data)])));
```

After the network request is done, we check that the third call to `dispatch()` sent us the same action object as a direct call to the `setData(data)` action creator.

Remember that we mocked the server response for the `axios` library with `mockRequest()`, passing it the stringified version of `data`.

Failed server access

The failing case is similar to the success one, except that we mock the server request to fail. There are two tests in this scenario, verifying that `API_FINISHED` was not dispatched and that `API_ERROR` was dispatched instead:

Failure tests framework

```
describe('error', () => {
  beforeEach(() => mockRequestError('recipes.json', 404, 'Not found'));

  it('should NOT dispatch API_FINISHED', () =>
    middleware(apiAction()).then(() =>
      expect(dispatchCalls[1][0].type).not.toBe(API_FINISHED)));

  it('should dispatch error', () =>
    middleware(apiAction()).then(() =>
      expect(dispatchCalls[1]).toMatchSnapshot()));
});
```

Here we have used all the methods discussed previously to test both cases.

Middleware Tests Summary

As our application grows, more complicated and asynchronous code will be moved to middleware. While this will cause the tests for the middleware to become complex, it will keep the complexity from spreading to other parts of our Redux application. The basic structure discussed here for the API middleware should be enough to cover most implementations.

We have left mocking `getState()` as an exercise for the reader. It is suggested that you take the sample project and modify the API middleware to read something from the state before the code that performs the API request (e.g., get an access token), and that you correctly update the tests to check that the store is accessed and correct values are used in the middleware.

Integration Tests

The role of the integration tests is to verify that all the parts of the application work correctly together. A comprehensive unit test suite will ensure all the reducers, action creators, middleware, and libraries are correct. With integration tests, we will try to run them together in a single test to check system-wide behavior.

As an example of an integration test, we will verify that when the `fetchRecipes()` action creator is dispatched, data is correctly fetched from the server and the state is updated. In this flow we will check that the API middleware is correctly set up, all the required action creators are correct, and the recipes reducer updates the state as needed.

Basic Setup

Since the integration tests will be using the real store, we can simply require and initialize it as in our regular application:

Integration test skeleton

```
import store from 'store';

describe('integration', () => {
  it('should fetch recipes from server', () => {
    // TODO
  });
});
```

Basic Integration Test

Our test will include four steps:

1. Verify the initial state.
2. Mock the data returned from the server.
3. Dispatch the action created by `fetchRecipes()`.
4. Verify that our state's `recipes` key holds the data returned from the server.

The full test looks like this:

Full integration test

```
import store from 'store';
import { fetchRecipes } from 'actions/recipes';
import { mockRequest } from 'axios';

describe('integration', () => {
  it('should fetch recipes from server', () => {
    const data = [{ title: 'test' }];

    expect(store.getState().recipes).toEqual([]);

    mockRequest('recipes.json', 200, JSON.stringify(data));

    return store.dispatch(fetchRecipes())
      .then(() => expect(store.getState().recipes).toEqual(data));
 });
});
```

To make sure our reducer updates the state, we first verify that our initial `recipes` list is empty and check that it was changed to contain the server-returned data after the `fetchRecipes()` action completed.

Integration Tests Summary

As can be seen from this simple test, doing integration tests in Redux is usually fairly straightforward. Since everything is driven by actions, in most cases our integration tests will follow the four steps outlined previously: we verify the initial state of the system, mock any external dependencies, dispatch an action, and verify that the state has changed and any external APIs were called as expected.

Summary

In this chapter we have discussed in detail various methods of testing Redux using the Jest library. Given the clear division of responsibilities in Redux and in keeping with its plain JavaScript objects and idempotent functions, most unit tests (and integration tests) are short and simple to write.

This in turn means that we, as developers, can minimize the time we spend writing tests and still have a comprehensive and understandable testing suite.

Congratulations, you have reached the end of this book! We hope it helped you understand Redux better. Keep reading if you are interested in the future of Redux and want to learn about Redux-related libraries and resources.

The Evolution of Redux

> A word from our friend Mark Erikson[73], Redux co-maintainer and author of the Redux FAQ and several of the Redux ecosystem resources linked to in the next section.

In many ways, Redux has been "done" since its initial 1.0 release in August 2015. The 2.0 and 3.0 releases less than a month later cleaned up a few bits of the public API, and 3.1 in January 2016 allowed passing an enhancer as the last argument to `createStore()`. The changes since then have mostly been small internal tweaks and minor cleanup that isn't meaningfully visible. Even the recent 4.0 release is primarily about updating TypeScript typings and smoothing out edge cases.

So where is Redux going in the future? There are a few different answers to that question.

Redux Roadmap

First, the core Redux library will remain stable, with some additional tweaks around the edges. For example, users frequently ask for `combineReducers()` to allow scenarios like passing additional arguments to slice reducers, or skipping warnings when state slices exist without a slice reducer to handle them. We may try breaking up the `combineReducers()` functionality into smaller functions to allow users to supply their own implementations for iterating over state values and combining them, or allow options to turn off warnings, then reimplement the internals of `combineReducers()` to use those customizable options. Overall, though, the core API will stay the same.

React-Redux, on the other hand, will see some major changes in the near future. React-Redux 5.0 was already a complete internal rewrite that moved the subscription logic out into memoized selector functions, and added top-down subscription behavior to ensure that parent components are updated before their children. With the release of React 16, the React team is pushing forward with changes to React's API and behavior that will require corresponding changes to the internals of React-Redux. In particular, the React team has proposed a replacement for the existing `context` API, which React-Redux currently uses to make the store instance accessible to nested components. The upcoming work on async rendering will also require changes and adaptation by all of the state management libraries in the React ecosystem. Look for more major changes to the internals of React-Redux over the course of 2018. Ideally, all of those changes will be handled in ways that keep the public API the same, so that your application code won't have to change.

If you'd like to get involved, there are always opportunities to help improve the Redux documentation. We've always got several open issues with the "docs" tag that are available for people to work on. There are also tentative plans to revamp the React-Redux docs, and possibly to rework the current build system for the Redux docs as well.

[73] https://twitter.com/acemarke

Growth of the Redux Ecosystem

The other big area of change is the growth of the Redux ecosystem. My React Boston 2017 presentation[74] covered some of the stats and trends showing how much Redux is used, and the wide variety of Redux-related addons. Some highlights:

- Redux is used in an estimated 55-60% of React apps, and Redux or Redux-inspired libraries are frequently used in the Angular, Vue, and Ember ecosystems as well.
- My Redux addons catalog[75] currently lists over 2,000 Redux-related addons, tools, and libraries.
- While there are many side effects libraries available, the four most popular are:
  - `redux-thunk` as a starting point for simple async logic
  - `redux-saga` for complex async logic using generators
  - `redux-observable` if you prefer to use observables
  - `redux-loop` for those who want to emulate Elm-style side effects in reducers
- There are many different approaches for fetching data and making network calls; everyone seems to be doing it differently.
- Store state persistence is very common.
- Redux-based routing is attracting interest as an alternative to React-based routing.

In addition, the Redux community continues to build valuable tools to solve a variety of use cases, from generating reusable action creators and reducers, to tools for data fetching and network behavior, to higher-level abstractions on top of Redux.

To me, the real excitement is what's happening in the ecosystem, and I can't wait to see what the community builds next!

[74] http://blog.isquaredsoftware.com/2017/09/presentation-might-need-redux-ecosystem/
[75] https://github.com/markerikson/redux-ecosystem-links

Further Reading

Despite its small size, the Redux library has had a huge effect on the way developers handle data management in single-page applications. It's already used in many large production applications, and a large ecosystem has grown up around it.

Today, there are a lot of great materials about Redux all over the internet. This book attempts to group together best practices for real-world use of Redux and show how to use it correctly in large applications—but since the web world is constantly moving forward, it is always a good idea to keep up to date and explore new libraries and methods. In this section of the book, we would like to mention some good Redux-related sources for further learning.

Courses and Tutorials

"Getting Started with Redux"[76] is a great free video course by Redux cocreator Dan Abramov, where he implements Redux from scratch and shows how to start using it with ReactJS.

Learn Redux[77] is a free video course by Wes Bos. You can follow along with the series to build a simple photo app using React and Redux.

Finally, the official Redux documentation[78] is a truly great resource: a well-maintained, constantly improving source of knowledge for this tiny yet powerful library.

Useful Libraries

The Redux ecosystem now includes dozens (if not hundreds) of useful libraries that complement or extend its features. Here's a short list of libraries that have gained widespread popularity and are strongly recommended for use in large Redux projects—we recommend that you check out the source code of these libraries to get a deeper understanding of the extensibility of Redux:

`reselect`[79]
: If you count the Redux store as a client-side database of your application, you can definitely count selectors as queries to that database. `reselect` allows you to create and manage composable and efficient selectors, which are crucial in any large application.

[76] https://egghead.io/courses/getting-started-with-redux

[77] https://learnredux.com/

[78] http://redux.js.org/

[79] https://github.com/reactjs/reselect

`redux-actions`[80]

Written by Redux cocreator Andrew Clark, this library can reduce the amount of boilerplate code you need to write when working with action creators and reducers. We find it particularly useful for writing reducers in a more ES2015 fashion, instead of using large `switch` statements.

`redux-undo`[81]

If you ever need to implement undo/redo/reset functionality in some parts of your application, we recommend using this awesome library. It provides a higher-order reducer that can relatively easily extend your existing reducers to support action history.

`redux-logger`[82]

`redux-logger` is a highly configurable middleware for logging Redux actions, including the state before and after the action, in the browser console. It should only be used in development. Our advice is to use the `collapsed: true` option to make the console output more readable and manageable.

`redux-localstorage`[83]

This reasonably simple store enhancer enables persisting and rehydrating parts of the store in the browser's `localStorage`. It does not support other storage implementations, such as `sessionStorage`. Be aware that `localStorage` isn't supported in private mode in some browsers, and always check its performance in large applications.

Resource Repositories

The Redux repository on GitHub has an Ecosystem documentation section[84] where you'll find a curated list of Redux-related tools and resources.

There is also a less-curated (and thus much larger) resource catalog managed by the community called Awesome Redux[85]. This repository contains a good amount of resources related to using Redux with different libraries such as Angular, Vue, Polymer, and others.

If you are looking for more React/Redux-focused material, Mark Erikson maintains a resource repository called React/Redux Links[86]. The materials there are separated by difficulty level, and a broad range of topics are covered (state management, performance, forms, and many others).

The same author also maintains a more Redux-focused resource list called Redux Ecosystem Links[87], which has a similar structure.

[80] https://github.com/reduxactions/redux-actions
[81] https://github.com/omnidan/redux-undo
[82] https://github.com/evgenyrodionov/redux-logger
[83] https://www.npmjs.com/package/redux-localstorage
[84] https://github.com/reactjs/redux/blob/master/docs/introduction/Ecosystem.md
[85] https://github.com/xgrommx/awesome-redux
[86] https://github.com/markerikson/react-redux-links
[87] https://github.com/markerikson/redux-ecosystem-links

Closing Words

Improving the Book

Writing this book was an amazing journey for us, and we really hope you enjoyed reading it.

This is the second edition of the book, improved mainly due to the great feedback we got from the Redux community.

If you find any issues, or have suggestions or comments on how to improve the book, we would really love to hear them.

Please drop us a line at *ideas@redux-book.com*.

What's Next

Done reading the book and still have questions, or wonder how the tools can best be used in large and complex Redux-based projects?

Feel free to reach out to us at *info@500tech.com*; we would love to try to help.