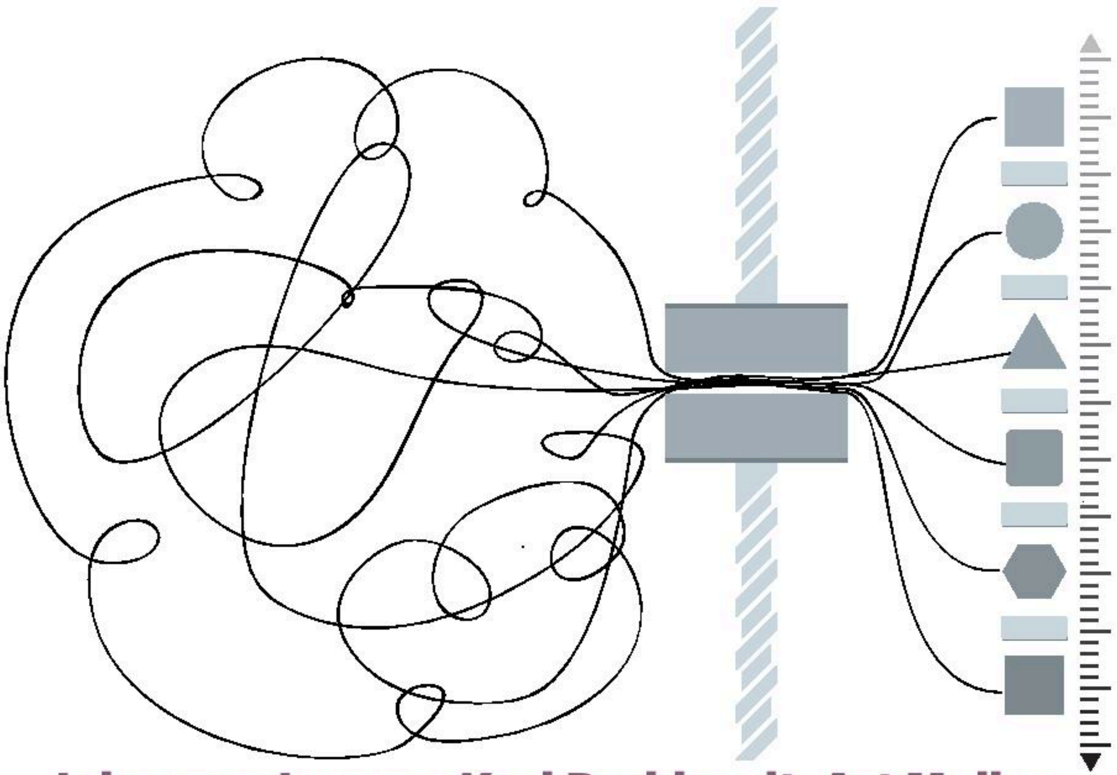
Bleeding Edge Press

Developing a Redux Edge



Johannes Lumpe, Karl Purkhardt, Art Muller, Darío Cravero, and Ezekiel Chentnik

Developing a Redux Edge

By Johannes Lumpe, Karl Purkhardt, Art Muller, Darío Cravero, and Ezekiel Chentnik

Developing a Redux Edge

By Johannes Lumpe, Karl Purkhardt, Art Muller, Darío Cravero, and Ezekiel Chentnik

ISBN 9781939902436

Published by: Bleeding Edge Press, Santa Rosa, CA 95404

Title: Developing a Redux Edge

Authors: Johannes Lumpe, Karl Purkhardt, Art Muller, Darío Cravero, and Ezekiel Chentnik

Acquisitions Editor: Christina Rudloff

Editor: Troy Mott

Cover Designer: Martin Murtonen

Website: bleedingedgepress.com

Table of Contents

Preface

Who is this book for?

This book is for anyone wanting to learn about Redux, a predictable state container for JavaScript apps. Whether you are looking to understand the basics, or learn the advanced concepts, there is something in this book for you.

If you're coming from React, you'll find additional goodies in this book. The sample application is built using React and features a lot of best practices. However, if you aren't using React, don't worry. The core concepts are agnostic to React and the sample application is simple enough that you can follow along easily enough.

Some of the chapters relating to the sample application are actually agnostic to React, such as Chapter 7, which covers Persisting State on the Client.

This book also contains a section on integrating Redux into an Angular 2 application. This section walks you through creating a small Redux-powered Angular 2 application that you can build and run.

What do you need to know prior to reading?

This book is aimed at intermediate developers who have a good understanding of creating single page applications with JavaScript. Having an understanding of ES6, functional programming, and React will certainly help too, but isn't necessary.

If you want to follow along with the examples you should also know your way around a terminal, and know what Node and npm are, as well as having them installed and ready to use.

Code samples

We provide a lot of code samples throughout the book, especially when we're building the sample application. We have put the sample application online so that you can download, run, inspect and compare it against your own application. You can view the code at: https://github.com/arturmuller/developing-a-redux-edge.

Author bios

The following authors contributed to this book.

Karl Purkhardt

Karl has been developing software for over 15 years, and started his career in programming by developing a modification for the popular game Half-Life. He has spent the last 10+ years working in web and is the author of redux-decorators. Karl brings a different angle to the book because he works with Redux outside of the React ecosystem.

Artur Muller

Art has been working with the web for over 10 years. Originally a designer, he's a self-taught programmer who loves building products. He has been working with Redux since its creation and has contributed to numerous open source projects including redux-keen and redux-ws.

Johannes Lumpe

Johannes is a full stack web developer and an active contributor to open source projects. He is the creator of redux-history-transitions, helped to build the React Native Playground, and was responsible for porting over the initial Redux Dev Tools to React Native.

Darío Cravero

Dario is a full stack developer with a background in Ruby. He's involved in a number of open source projects including redux-pouch and bublé. Dario enjoys working on experimental projects and is currently composing a more functional web through one of his open source projects called Panels.

Ezekiel Chentnik

Zeek is a Principal JavaScript Engineer with Northwestern Mutual (Financial Tech) and a CTO at Gufsky. His experience spans global markets (Asia, Europe, Americas) with a passion for large scale e-commerce, financial tech, mobile, progressive web applications, and full stack JavaScript development.

Technical reviewers

We would like to the thank the following technical reviewers for their early feedback and careful critiques.

Mark Erikson is a Redux expert. He wrote the official FAQ page for the Redux docs, and was given commit access to the Redux Github org as a result. Mark has assisted with answering questions and managing the docs.

Erik Rasmussen is known to the Redux community for his **react-redux-universal-hot-example** repo, for creating the **Ducks** specification, and for creating the very popular **Redux Form** library. He has been using Redux since its invention, even before Dan's 2015 React Europe talk.

Sven Anders Robbestad is a a front-end consultant currently working for Acando in Norway. He has been working with JavaScript since it was introduced in the mid-90s and has worked for a number of large corporations in Norway since then. He has also given talks on JavaScript and front-end development in Norway and abroad and has published a book on writing ReactJS applications.

Simen Bekkhus has been a front-end developer for 3.5 years, and has worked on projects using Backbone, where he stood for the migration to React and Redux, and in his new job, on an universal React and Redux app. He actively contributed to OSS, especially geared towards Webpack and Redux.

Michael Baldwin is a software engineer currently building an application with Redux at a startup called Notion AI. He is a recent graduate from **Grand Valley State University** with a Bachelor of Science in Computer Science.

Why Use Redux? 1

State is the heart of all rich applications, and as application developers we spend a lot of time managing state within our applications. Without state a web application is nothing more than a static HTML page. If we want to build feature rich web applications we must become proficient at managing state.

Managing state is problematic

As web technologies have matured, web applications have become more complex and feature rich, pushing the boundaries of what's possible. These feature-rich web applications come at a cost, and that cost is complexity. A large piece of that complexity is the management of application state.

Managing this complexity and the extra state required to enable these rich features is hard. Traditionally, we have used two-way data binding to manage state, allowing state to be automatically updated by binding it to the view layer. This is the approach a lot of frameworks took in the early days of web applications, and it's still in use today.

This works out fine for smaller applications, but has proven to be difficult as the complexity of web applications have increased, often resulting in opaque systems where data flow and state changes are non-deterministic and hard to debug. As more features are implemented into these troublesome architectures, the issue is compounded, creating a vicious circle.

Part of this complexity is a result of mixing state mutation and asynchronicity, two concepts that are inherently difficult to understand. In Redux circles this problem is referred to as Mentos and Coke. The point being that these concepts can be unpredictable and messy when mixed.

How Redux helps

Redux aims to decouple state mutation and asynchronicity, separating them so that you can reason about them individually. If you have a problem with state, you can debug the state tree to determine how the state was created without having to worry about asynchronous logic. Likewise, if you have a problem with your asynchronous logic, you can focus on debugging it without having to worry about state.

Not only is this better for your sanity, it opens up a range of additional benefits, ranging from improvements to developer workflows to easier unit testing. All of this due to the separation of asynchronous logic from state mutation.

Redux does more than simply separate these concerns, it also implements restrictions on how state can be mutated, making state mutations predictable, and the state of your application much easier to reason about.

Redux wants to give you control back, it wants you to enjoy programming again, and it’s going to do this by providing you with an architecture that is developer friendly, predictable, and robust.

Redux background

Redux was created by Dan Abramov, who is currently a developer working on React at Facebook. Dan created Redux while working on a talk called “Hot Reloading with Time Travel for React Europe 2015.” In order to demonstrate these two concepts, a predictable architecture that used a single state tree was required. It was this desire that drove the creation of Redux.

The birth of Redux

Dan initially tried to use traditional Flux, but he had many difficulties. For example, in order to hot reload store logic, a store had to be a pure function. This wasn’t the case with traditional Flux stores that have behavior such as subscribe and register built in. As you remove this behavior from traditional Flux stores, you end up with stateless stores, and the first step towards Redux.

Note: In Redux we call these stateless stores reducers, and we’ll refer to them as that from now on.

Once you have stateless stores, which we’re now calling reducers, you can hot reload the state logic. These reducers were still tied to a dispatcher though. So, instead of having multiple stateless reducers, a single reducer was introduced. This single reducer removed the need for a dispatcher because actions then have a single destination.

The obvious problem with having a single reducer is that there is no separation of responsibilities. This is easily overcome by the fact that reducers are just pure functions, and that means they can be composed of other functions. So a single reducer can be composed of many smaller reducer functions, each one with a particular responsibility.

At this point you essentially have Redux. You have a single store and a single root reducer created from a pure function. These are two of the Redux core principles. This early implementation has been improved and polished, becoming the library that we know today.

Redux influences

Let's examine the major Redux influencers.

Elm

Redux produces an architecture that is very similar to the architecture used in Elm - a functional programming language for HTML apps. It can even be considered an almost direct translation into JavaScript. While this is true, Redux is not simply a direct copy of Elm. Elm is more of an indirect influence, and the fact that they are very similar is proof that great architectures naturally arise from similar design goals.

ES6

ES6 is heavily used in Redux. You will benefit greatly from being comfortable reading and writing ES6 code when learning Redux. Redux doesn't use ES6 just for the sake of it, the language features most used actually make writing Redux application code much easier.

We're not saying that you need to stop and learn ES6 inside-out before continuing, rather that you should be comfortable with it, because you're going to see a lot of it, and you'll pick things up much quicker if you can focus on learning one thing at a time.

At the very least, you should be comfortable reading arrow syntax functions, and have a good understanding of the `const` and `let` keywords, as well as the spread operators for arrays and objects. Object spreads are currently an in-progress proposal for the JavaScript language, and widely available through tools like babel, and useful when writing reducer logic.

Functional programming

Redux embraces functional programming and incorporates many functional programming concepts into its methodology. Although functional programming is not a prerequisite to learning Redux, knowing about the core concepts of functional programming will certainly help reduce the initial learning curve.

Immutability

Immutability is a related concept to functional programming. Immutable data is simply data that cannot be changed after it has been created. Redux embraces immutability, since it's at the core of at least one of the three principles of redux: state must be read only.

Redux doesn't enforce this, and you're free to use mutable or immutable objects to model your state, but you should follow the rules of immutability and ensure that you don't mutate state to benefit from the predicability that it provides. This also ensures that scenarios like time travel debugging will work correctly, and that UI components will update properly.

Flux

Redux is inspired by Flux - an application architecture for building user interfaces that uses a unidirectional data flow. If you compare the two you will notice similarities - both help to manage application state, and both implement mechanisms for managing how changes are made to the state.

Although similar, there are some key differences between the two. One key difference is that in Flux there is the concept of a dispatcher. A dispatcher is responsible for relaying actions to listening stores. Redux, however, eliminates the need for a dispatcher through the use of a single reducer, pure functions, and function composition.

Another key difference is that Flux has many stores with each store managing a separate piece of state for a particular area of the application. In Redux there is only one store and one single source of state - referred to as the single source of truth. You can still manage state for different areas of the application, but in Redux you use reducers, and function composition, to achieve this.

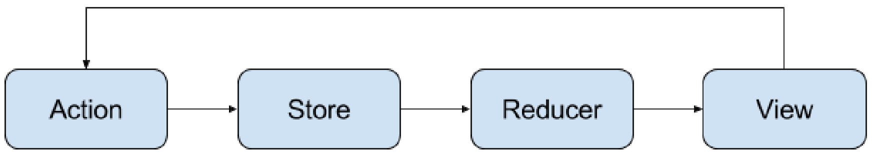
Both approaches solve the same problem, they just do so it in slightly different ways.

Unidirectional data flow

With roots in functional reactive programming, unidirectional data flow has gained popularity in recent years. The main principle is that data flows through your application in a single direction, making the flow of data through your application much easier to reason about.

With unidirectional data flow architectures like Flux and Redux, your state is contained in stores (a single store, in Redux) and updates to the state are managed through actions that describe how the state should be transformed. The view layer can then subscribe to the store(s) to receive new state whenever it's changed.

The following diagram illustrates the typical data flow in a Redux application. Here you can see that an action is dispatched to the store, so the store then passes the existing state and the action to the reducer, which transforms the state. The new state then flows through to any listeners, such as views. A view can dispatch further actions, starting the process again.



With data flowing in one direction to your views, they become functions of your application state, making them much more predictable. You can give the views the same data and you will get the same result, which is similar to a pure function.

Two way data binding

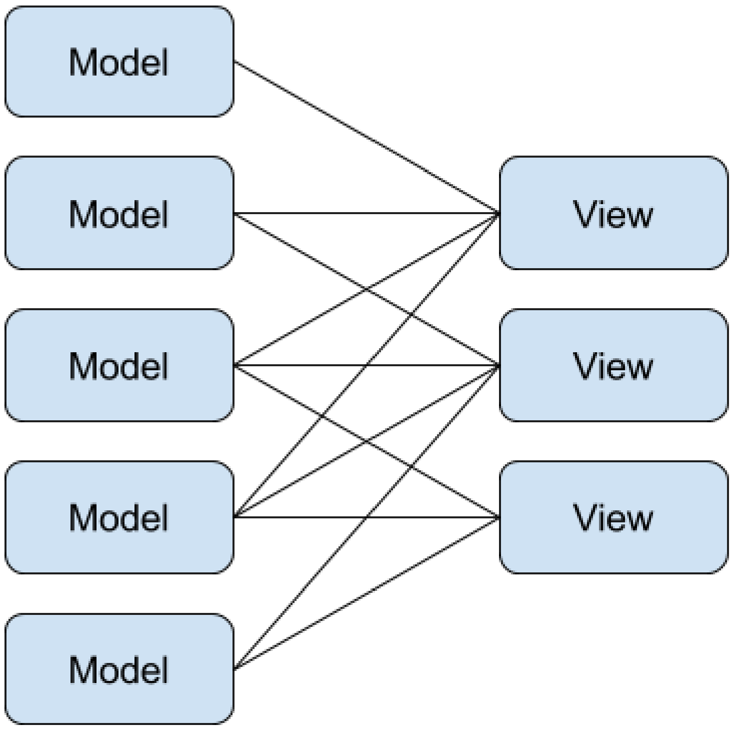
Another approach to managing data flow is two way data binding or bidirectional data flow. This has been a common approach for many years, and while it can work well for some applications, it does have known problems, such as cascading updates. A cascading update is where a change to state in one model can result in unpredictable changes to state throughout the application.

Two way data binding works by syncing the data between a model and a view. This is where the name two way data binding comes from, since the data is bound to both the model and the view, and can flow in either direction.

Changes to the model are propagated to the view and vice versa. For example, entering text into an input box in the view can result in a direct modification of data on the model. This is discouraged in a unidirectional architecture, and changes are never directly made to state.

Another major problem with two way data binding is that it results in complicated and unpredictable data flow. When state can be mutated by multiple sources, it's difficult to reproduce and debug issues involving changes to that state.

The following diagram illustrates the problem with two way data binding.



In the above diagram you can see that the dependencies between models and views have become incomprehensible. For the majority of models, it is no longer clear which views are mutating its state, so debugging issues relating to state becomes difficult. This is just a simple contrived example. The problem is usually much worse in large applications where the number of models and views are usually in double digits.

The three principles

The whole concept of Redux revolves around three core principles.

- Single source of truth
- State is read only
- Changes are made with pure functions

Let’s look at each one in isolation to understand their value.

Single source of truth

The first principle of Redux is that there is a single source of truth for all state within your application. This single source of truth is a state tree, which is simply a container for the state, which is usually a plain old JavaScript object.

```
const state = {
  todos: []
};
```

This state tree is managed by another component called a store, which we'll look at later. In Redux there is a single store, and therefore a single location for all of our state.

This is opposite to the approach that most traditional web applications take, where state is scattered throughout the application. For example, in a typical MVC application, state is stored in models, collections, and views.

There are some nice advantages to a single source of truth. For one, it's much easier to reason about state that is stored in a single location. It's better for your sanity, but there are other, not-so-obvious, benefits too.

One not-so-obvious benefit is that a single state tree offers us the ability to create universal applications much easier than with other architectures. This is due to the state being in a single location, making it easy to serialize and share between the client and server.

Another not-so-obvious benefit is an improved development experience. Similar to enabling easier universal applications, a single state tree also enables you to easily persist the whole of your application state. It is simply a case of saving the state (e.g. to local storage) and loading it again on page load.

State is read only

The second principle of Redux is that state is read only. This doesn't mean that you can't change the state of the application, it means that you should never modify the state tree directly outside of the store.

Instead, modifications to the state tree should be made by dispatching actions. An action is simply an object that describes the change you want to make to the state. They are dispatched with a simple method call, similar to dispatching an event with an event emitter.

Following this principle provides an architecture where changes to state are centralized and applied, one at a time, in a strict order. Like state being read only, this also provides some not-so-obvious benefits.

One such benefit is the ability to rollback, commit, and replay actions to reproduce state, a concept called time travel debugging. It also helps to remove hard-to-debug race conditions where state has been changed in multiple locations in a non-deterministic way, which is a common problem in MVC architectures.

Centralizing changes to state is a different approach to your typical MVC architecture, where state changes can come from multiple places. For example, model state can be changed by the model itself, the model's collection, or a view that presents the model.

Changes are made with pure functions

The last Redux principle states that changes are made with pure functions. What this means is that the state tree, the single source of truth, can only be changed by passing it to a pure function that then applies the change and returns the new state.

A pure function is simply a function that provides the same output given the same input and has no side effects. In other words, a pure function does not change any external state or interact with the outside world in any way.

In Redux, these pure functions are called reducers. They are called reducers because they follow the same pattern as reducers from functional programming. This means they accept an accumulator and a value and return a new single value, for example the `Array.reduce()` method. The following is an example of a Redux reducer that exhibits these behaviors.

```
function reducer(state = 0, action) {
  switch (action.type) {
    case 'multiply':
      return state * state;
    default:
      return state;
  }
}
```

The main advantages of using pure functions to manage changes to state are predictability and testability. They are predictable because you always get the same output when called with the same input. This also makes them easy to test, as does the fact they don't interact with the outside world, which results in less mocking and stubbing of external dependencies.

With these benefits in mind, and the fact that there is only a single source of truth, you can see how Redux encourages an application architecture where the core of your state management is predictable and easy to test. Compare this to other approaches, even unidirectional approaches such as Flux. In a Flux architecture state does not possess the qualities of predictability and testability, instead the the opposite is usually true.

And because managing state is such a large part of what you do when developing rich web applications, you benefit greatly from a data flow architecture that provides predictable state that is easy to test. This predictability gives you confidence and control, and the ability to easily test how state changes allows you to verify that your predictions about state transformations are correct.

Looking ahead

We've discussed what Redux is, what problems it solves, and what inspired its methodology. In the next chapter we're going to look at the core concepts that define Redux. As we progress through the book we're going to build on these core concepts, developing our own Redux application using React for the view layer.

While Redux can be used with other frameworks, it has close ties to React, which has a big community, a lot of support, and many tools available for this particular stack. Support for using Redux with other frameworks is improving, but due to how closely tied Redux and React are, it's unlikely that you'll find the same level of support elsewhere.

However, with that said, we will be showing you how to integrate Redux with two of the biggest technologies in use today: React and Angular 2. We'll look at these integration paths before we start on our sample application.

Conclusion

In this chapter we looked at where Redux came from, its inspirations and the reason for its creation. We also looked at some of the issues Redux attempts to address, learning about both uni directional architectures and two way data binding. Finally, we took a look at the three Redux core principles, which we need to keep in mind as we continue on our path to gain a Redux edge.

Core Concepts 2

Redux is based on a few simple core concepts, such as actions, reducers, store and middleware. In this chapter we will examine these concepts in detail, to better understand how they form the Redux system. As we go along we will also learn how to test each of these parts to make sure that our system works as expected.

Actions

Let's start with actions, which are used to communicate between an application and Redux.

What are actions used for?

Actions are used to tell Redux that something in an application has changed, or that the user has interacted with the application in some way. Actions are descriptor objects that contain information about what exactly changed along with potentially some additional data associated with that change.

The shape of an action

Actions in Redux are plain objects with a `type` property. Nothing else is enforced by the library. Apart from that you are free to add whatever properties you'd like. Here are two basic example actions:

```
{
  type: 'CHARACTER_TYPED',
  char: 'a',
}

{
  type: 'BACKSPACE',
}
```

Notice how the CHARACTER_TYPED action has a `char` property but BACKSPACE only has its `type`? Not every action needs data associated with it. Sometimes the `type` property already conveys enough information about the action that should happen.

Note: there has been a debate about whether actions should describe what we want the application to do or just what happened, e.g. REFRESH_USER vs REFRESH_BUTTON_CLICKED. It is really up to you to decide which style better fits your project - choose whatever you are comfortable with.

Action creators

Manually writing action object literals seems tedious and error prone. Apart from that, if something were to change for those actions, it can be a maintainability nightmare. From a testability perspective, it is easy to see why just writing plain object literals is a bad idea. Action creators can help us.

Action creators in their most basic form are nothing more than plain functions that return action object literals for us. This helps prevent us, though, from making typing mistakes when using the same action multiple times. Action creators also help keep things DRY, because actions are neatly tucked away inside a function instead of being spread throughout the codebase as object literals if you use the same actions in multiple places. Here is how a basic action creator might look:

```
// ES5
function insertCharacter(char) {
  return {
    type: 'CHARACTER_TYPED',
    char: char,
  };
}

// ES6
const insertCharacter = char => ({
  type: 'CHARACTER_TYPED',
  char,
});
```

The benefits of this should be immediately visible: there is no way to make typing mistakes anymore, and even if we did mistype the action type, we'd only have to fix it in one place. The code where you use this action is less verbose.

Although a great improvement, this is not the only thing action creators are useful for. Since they are functions, they can basically do any kind of work before returning an action object. They can be used for asynchronous actions, for example.

Maintaining your action shape

Right now we have a loosely defined structure for our actions and the only required property is `type`. They could potentially contain whatever keys we'd like to add to them. But instead of going wild and adding a lot of random properties, it is considered good practice to use a well-defined structure for our actions. There is a recommended way of creating actions called **Flux Standard Action**, which can be very useful to keep one's actions clean and well-formed, but it is up to you to determine what kind of structure you want to use for your actions.

Actions need an interpreter

On their own these actions don't do much. When combined with a reducer that knows how to interpret them, they become useful building blocks for applications. It is important to note that in Redux, the only way to modify your data is through actions. The actions become a middleman between your application and Redux.

Reducers

Now that we know what actions and action creators are, we are ready to move on to reducers, because actions alone are not very useful and they cannot function on their own. In order for them to be useful, they need to be processed/interpreted by a reducer.

What are reducers?

A reducer is a function that takes in inputs and returns some new output:

```
(a, b) => c
```

The important thing to grasp here is that a reducer is just a basic function that operates on its inputs to produce some output - it reduces the two arguments into a new result. You might have already worked with such a function while reducing an array:

```
const result = [1, 2, 3].reduce((sum, value) => sum + value, 10);
// result = 16
```

Because we want our results to be deterministic and reliable, the reducer function must be pure. Being pure means that it only relies on its input to produce some output. It must not reference any mutable variables outside of its own scope and will always produce the same output for the same set of inputs. In addition to that it must not use any kind of non-deterministic functions like `Date.now()` or `Math.random()`, since the results of those functions will change each time they are called and are not tied to the input they receive. Think of it as a mathematical function: it only computes something and it does not do anything else. This is against the rules of reducer behavior.

Let's illustrate this with an example of how a basic reducer would be tested:

```
import test from 'tape'

const reducer = (a, b) => a + b;

test('Basic reducer', t => {
  t.equal(reducer(1, 0), 1);
  t.equal(reducer(21, 21), 42);

  const result = reducer(41, 1);
  t.equal(reducer(result, 42), 84);

  t.end();
});
```

This test will pass just fine, because our reducer is completely pure and predictable. If we were to update this example and change our reducer code to the following, the tests would instantly break:

```
const reducer = (a, b) => a + Math.random()
```

As simple as this example is, it showcases a very significant issue: we cannot in any way determine what the correct output of the function should be. We know that it should be the first argument plus some number. But we have no idea what the next value would be. This makes our reducer impossible to test and non-deterministic. Every piece of non-deterministic code should live outside the reducer, so that the reducer itself can be kept pure.

Using reducers in Redux

In Redux, reducers are used to facilitate state transformations based on actions. We already know what a reducer is, so let's re-label the above reducer function's arguments and return value with Redux terminology:

```
(state, action) => state
```

It receives `state` and `action` and computes a new `state`. Here is a reducer which can process the `CHARACTER_TYPED` and `BACKSPACE` actions from the previous section about actions:

```
const reducer = (state = '', action) => {
  switch (action.type) {
    case 'CHARACTER_TYPED':
      const { char } = action;
      return state + char;

    case 'BACKSPACE':
      return state.substr(0, state.length - 1);

    default:
      return state;
  }
}
```

Our reducer in the above example is a good example of a pure reducer. It does not reference anything outside its own scope and does not make use of any non-deterministic function. We have to reiterate how important it is to keep reducers pure, so that we can easily test and and make sure that we always receive the same output for the same input. Since this reducer looks very different from the basic number-adding example, let's walk through it.

First of all, we have to make sure that we have a valid `state`. Redux will initially pass `undefined` for the `state` argument to our reducers so that they initialize themselves. This will only ever happen once and is the perfect time to set an initial state. We do this by using ES6 default arguments. Now the `state` will become an empty string whenever we receive an `undefined` state value.

In the action section earlier in this chapter we learned that an action always needs a `type` property, so we can use it in combination with a `switch` statement to determine which action we received.

When receiving an action with the CHARACTER_TYPED action type, we extract the `char` property from this action, concatenate it with the previous `state`, and return the result as a new state.

Likewise, when we receive a BACKSPACE action, we return the current state minus the last character.

It is important to note that whenever a reducer does not know how to process a type of action, it has to return the previous state unmodified, hence the `default` case. This has to be done because in a real application there will be multiple reducers and multiple actions. If a reducer only returns a result if it recognizes the action, an unknown action effectively resets its `state`, and you would have major problems across your application with state being reset randomly everywhere. A reducer must handle an action and return a new state or just return the passed in state unmodified.

Reducers are composable

Right now we only have a single reducer. In fact, we will only ever have a single reducer function. But that does not mean that the reducer can't be composed of multiple reducers. Reducer composition is a powerful tool to refactor big reducers into smaller components, and to organize state into separate slices.

It is very easy, however, to imagine how the reducer function will get massive if it has to handle more than a few actions, especially if each of those actions requires more than a one-liner to get processed.

There are two common ways to make sure that this does not get out of hand. The first common pattern to keep things tidy is to delegate the processing for certain actions to smaller reducer functions. Instead of handling the state merging for CHARACTER_TYPED directly in the main reducer, we just delegate to a separate one:

```
const appendChar = (state, action) => state + action.char;

const reducer = (state = '', action) => {
  switch (action.type) {
    case 'CHARACTER_TYPED':
      return appendChar(state, action);

    case 'BACKSPACE':
      return state.substr(0, state.length - 1);

    default:
      return state;
  }
}
```

Granted, this example is a bit contrived and you might think that this is overkill. For this tiny function you might be correct, but if you keep a larger real-world application in mind, it's easy to see how this can greatly benefit the codebase.

This is one way to keep the reducer neat and tidy, so that we won't have to wade through hundreds of lines of code to understand how certain actions are being handled.

But what if we want to separate our state into separate slices, which are oblivious of each other's existence? For the sake of argument, let's say that our application allows users to select a theme and choose whether to activate a mode for colorblind people. That state clearly does not belong together with the domain/entity state. This would be UI state and thus it should live in a separate reducer. So how would we go about this? It's really straight forward:

```
const editorReducer = (state, action) => { /* omitted for brevity */ };
const uiReducer = (state, action) => { /* omitted for brevity */ };

const reducer = (state = {}, action) => ({
  ui: uiReducer(state.ui, action),
  editor: editorReducer(state.editor, action),
});
```

As you can see above, all you have to do is create a new reducer function that selectively passes slices of our state atom to sub-reducers. This is a very powerful thing, because it helps to decouple the reducers from each other and we need to think about how we want to split up our state. This can greatly help state organization, but it not something that should be done just for the sake of doing it. The boundaries we define have to make sense or else we will not gain any benefits from it. Even worse, this could negatively impact our application state. As someone once said, “With great power comes great responsibility.” So be sure you use this technique wisely.

In the above example everything is manual and it can get tedious really fast if we add more slices. Fortunately, we have a tool at our disposal to help us!

COMBINEREDUCERS

Redux ships with a handy helper called `combineReducers`, which does the same thing we did in the example above. In addition, however, the combineReducers helper function turns an object whose values are different reducing functions into a single reducing function that you can pass to createStore.

> The resulting reducer calls every child reducer, and gathers their results into a single state object. The shape of the state object matches the keys of the passed reducers.

Here is how the above example looks when using the helper:

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

const editor = (state, action) => { /* omitted for brevity */ };
const ui = (state, action) => { /* omitted for brevity */ };

const reducer = combineReducers({
  ui,
  editor,
});
```

We renamed the reducers so we can just add them using short object notation. The keys of the provided object will become the names of the state slices for each reducer, so the UI state will be mounted under `state.ui`, and the notes state under `state.notes`.

`combineReducers` not only automatically executes our reducers with the correct state slice and action. It also asserts that the reducers return a valid state when the initial state is `undefined`, or when an unknown action is passed to them. This does not mean that we shouldn’t test our reducers for those cases, but it is a nice safety net to have.

The tool only adds keys from the passed-in object that contain a function, so there won't be errors like `Uncaught TypeError: x is not a function` because it accidentally tried to execute a non-function type as a function. This is useful to know, because it could leave us clueless as to why a certain state slice isn't mounted or a reducer isn't executed. In these cases the first thing to do is to check whether the provided reducers are all actual functions or whether we accidentally provided an object, or something else.

Note that even though you will often see this helper used at the root of the state tree, there is nothing stopping you from using it on its nested parts too!

It is important to understand though, that `combineReducers` is nothing more than a helper, which might or might not fit your needs. You can use a completely custom version if you want to. For example, you might want to access the global reducer state in all of your reducers. You can implement this as a custom version of `combineReducers`, which injects the complete state to each of the reducers as a last argument. The possibilities here are endless, and one size does not always fit all, so use what's appropriate. It's all just functions!

Store

Until now we had to pass state and actions manually to our root reducer. Not only is this cumbersome, but it isn't very useful, because we don't get notified about our state changes. The store changes this.

The store in Redux is what coordinates passing actions to your reducer, notifying you of changes to your state. It encapsulates the state and offers an interface that you can use to interact with it. This interface is comprised of the following:

- `dispatch(action)`
- `getState()`
- `subscribe(listener)`

We will walk through each of these methods, but first let's create a store, otherwise we'll have nothing to call these methods on:

```
import { createStore } from 'redux';
// reducer is the root reducer we defined when talking about reducers
const store = createStore(reducer);
```

This creates a basic store. As soon a store is created, Redux will initialize the reducer with an `undefined` state as mentioned before. Our store is now ready for prime time.

With our new store, we can now dispatch actions. Dispatching is the act of passing an action to your store, which in turn then passes to the root reducer for processing:

```
// The following are all equivalent. We will stick with the inline usage but
show these once for completeness
```

```
// literal
store.dispatch({
  type: 'CHARACTER_TYPED',
  char: char,
});

// stored action creator result
const actionCreatorResult = insertCharacter('a');
store.dispatch(actionCreatorResult);

// inline usage of action creator
store.dispatch(insertCharacter('a'));
```

This will trigger a computation in the reducer and save the new state in the store. We can confirm this by using `getState()`:

```
const state = store.getState()
console.log(state);
// logs 'a'
```

So we know that the store relays the action to the root reducer and we can get the new state using `getState()`. This is a start, but we are missing an important ingredient. In order to update anything we need to know when an action was dispatched. Enter `subscribe`.

With `subscribe` we can add a listener to the store, which gets triggered every time our state changes. So now we can - from anywhere in our application - receive a notification that our state changed. That's great and here is how that looks:

```
const unsubscribe = store.subscribe(() => console.log(store.getState()));
```

If we now dispatch an action, the listener will log the new state to the console. Actions can now be dispatched to the store at will from everywhere in our application, and whenever the state changes we will be notified and can react to the change:

```
store.dispatch(insertCharacter('a'));
// logs 'a'

store.dispatch(insertCharacter('b'));
// logs 'ab'

// `removeCharacter` is an action creator wrapping our `BACKSPACE` action
store.dispatch(removeCharacter());
// logs 'a'
store.dispatch(removeCharacter());
// logs ''
```

If we are not interested in receiving store updates anymore, we can call the `unsubscribe` function, which was returned when we `subscribed` to the store and then remove ourselves as listeners:

```
unsubscribe();

store.dispatch(insertCharacter('a'));
// nothing is logged
```

We now have a fully functional, basic store instance.

Middleware

Redux is already very useful on its own, because it allows you to update your application state in a well-defined way, and makes sure that as long as you respect the rules, your state is always going to be deterministic and relatively easy to debug. But there is more you can do. With middleware, Redux offers a powerful plugin-system, which can be used to hook into the action dispatching process.

Why does middleware exist?

Since Redux was built with the developer in mind, it does not come as a surprise that the original thought behind middleware was driven by providing a better developer experience. Needing to log actions, catch crashes, and perform these actions separately lead to the concept of middleware in Redux.

Middleware acts as a pipeline that all dispatched actions go through before reaching the store. Each middleware can inspect actions, transform actions, pass them on, or even stop them from reaching the store. But how does this work? In order to properly use middleware it is important that you understand its inner workings.

Anatomy of a middleware

A piece of Redux middleware is just a higher order function and it looks like this:

```
// ES5
function (store) {
  return function (next) {
    return function (action) {
      /* ... */
    };
  };
}
```

```
// ES6
store => next => action => { /* ... */ }
```

Middleware then is a function that returns a function, that returns another function. This is due to the way Redux's `applyMiddleware` helper - which we will talk about in a bit - composes the middleware. First, our middleware receives a `store` argument that is not the actual store, but a subset of the store's API, containing the `dispatch` and `getState` functions. The `next` argument is the next middleware in the chain, which is needed so we can pass on the action. And `action` is the actual action that has been dispatched.

When dispatching an action, it will be piped through the complete middleware chain before it reaches the original `dispatch` function of the store. This has some very important implications. We are in full control regarding an action's fate. We can transform it however we'd like to, and we can even completely abort the dispatch process by not forwarding it to the next middleware in the chain. This enables you to implement things like asynchronous actions and other nice enhancements.

How to use middleware

In order to see how easy it is to create and use middleware, we will implement a little logger middleware. It will log the state before our action, then pass the action through to our next middleware, and finally log the state after our action has been processed. Here is a very basic implementation:

```
const loggerMiddleware = ({ dispatch, getState }) => next => action => {
  const prevState = getState();
  const result = next(action);
  const nextState = getState();

  console.group('Dispatch', new Date());
  console.log('Previous state', prevState);
  console.log('Action', action);
  console.log('Next state', nextState);
  console.groupEnd('logger');

  return result;
}
```

And that is it. This is a middleware that will log every single action that passes through it to a custom group in the console. It will show the previous state and the next state, so you can see what changed. This is of course a very basic implementation, but you can go further and show the actual diff between the previous and the next state. But it shows how easy it is to create our own useful middleware. After logging, we return the result of our call to `next`. A middleware should always return a result, whether it is the result of the next middleware in line or a custom one. If nothing is returned, then there is no way to use the result of the dispatch. This will become especially important when we talk about async actions.

Now, you might wonder why we used `next` and not `dispatch` to forward the action. The usage of `next` and `dispatch` depends on the middleware and the transformation that is applied to the action. `next` will pass the action to the next middleware in line and eventually to the store. `dispatch`, on the other hand, will send back the beginning of the middleware chain. In the above case, we are not transforming our action in any way. We are just passively logging some data. And we have no condition in place to determine whether we want to send the action back to the beginning of the chain or let it continue its way normally. If we were to use `dispatch`, we would cause an infinite loop and a stack overflow. It is important to repeat this: we only use `dispatch` if we have a way to distinguish between actions we want to process, and actions we just want to pipe through to the next middleware in line.

We have a logger middleware, so let's actually use it:

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

import reducer from './path/to/our/reducer';
import logger from './path/to/our/loggerMiddleware';

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

If we now call `store.dispatch()` with an action, it will get properly logged to the console using our middleware, including the previous and next state! We have learned about the concept of middleware, what it is used for and how to create and use a basic middleware. Knowing all of this, we can now look at store enhancers.

Store enhancers

Store enhancers are the big brother of middleware. They are used for things we cannot achieve with middleware. Store enhancers wrap around the existing store, providing their own versions of the store API functions to implement special behavior. One example is time-travel debugging, as mentioned in the Redux documentation. Another example, `applyMiddleware`, is actually a store enhancer, which swaps the store's dispatch method with a custom one that allows it to iterate through all of the used middleware.

The signature of a store enhancer looks like this, straight from the docs:

```
type StoreEnhancer = (next: StoreCreator) => StoreCreator
```

And `StoreCreator` looks like this:

```
type StoreCreator = (reducer: Reducer, initialState: ?State) => Store
```

In JavaScript, the above would translate to:

```
const enhancer = createStore => (reducer, preloadedState, enhancer) => {
  const store = createStore(reducer, preloadedState, enhancer);
  const { dispatch } = store;

  const newDispatch = action =>{
    const begin = Date.now();
    const result = dispatch(action);
    const end = Date.now();
    console.log(`Dispatch took ${end - begin}ms`);
    return result;
  }

  return {
    ...store,
    dispatch: newDispatch,
  };
}
```

Between creating and returning the store, we have the ability to wrap the store methods and return a new store object. The above is a very silly store enhancer, which tracks the dispatch duration. This can also be implemented as middleware with the caveat that the middleware that measures this might perform some additional, expensive work after this dispatch occurred and that might not be taken into account.

The above example is very basic, but it shows that store enhancers are a lower level mechanism when compared to middleware. Almost everything that is needed for a normal application can be created using middleware, so 99% of the time we have no need to actually create a custom store enhancer. It is good to know about them though, in case you ever feel the need to do something extraordinary. When that time comes, a good rule is to first check whether an idea can be implemented as middleware, and if not, only then to resort to using a store enhancer.

How does Redux work?

Redux's creator, Dan Abramov, described it as "A change emitter holding a value."

Is it indeed that simple?

In this section we will dive deeper into what makes Redux work, while reconstructing a simplified version of it with the intention of demystifying it and helping our readers sink in its internals better.

Redux's main public API consists of three methods: `getState`, `dispatch` and `subscribe`. There is a fourth method called `replaceReducer`, but it's meant for advanced use cases -such as code splitting- which we don't need to worry about for now.

Let's go ahead and tackle them one by one!

getState()

Since Redux is supposed to hold a value, we will need a place to store it and a way to retrieve it.

```
// Define a variable to hold that value which we'll call: state
let state;

// Expose a way to access it
export function getState() {
  return state;
}
```

dispatch(action)

After storing our state, we need to be able to change it. As you know, Redux works under the premise that it won't let a user alter the state directly, instead you describe how the state should change through an action.

```
export function dispatch(action) {
  // Set the state to whatever the reducer returned after being called with
the current state and
  // the action we want to process.
  state = reducer(state, action);

  // As we will see in future chapters, it is handy to return the action it-
self as a user could
  // leverage it to, e.g., wait on a Promise's result to do further tasks.
  return action;
}
```

The `reducer` function we are calling inside `dispatch` is defined by the user. The real library lets you define this when you call `createStore` **http://redux.js.org/docs/api/createStore.html**.

subscribe(listener)

The last piece of the puzzle gives us the ability to react to our state changing.

```
// As many actors might be interested in this piece of state, we will define
a place to hold
// our list of listeners
let listeners = [];

export function subscribe(listener) {
  // When someone subscribes we store the listener on the list...
  listeners.push(listener);

  // ...and give them a function that will allow them to stop listening when-
ever they think they
  // don't need it anymore.
  return function unsubscribe() {
    listeners = listeners.filter(l => l !== listener);
  };
}
```

There's one thing missing for our implementation to fully work: we actually need to call those listeners when the state changes! Since our state changes when an action is dispatched, `dispatch` seems like a good candidate. Let's modify it to fit our needs:

```
function dispatch(action) {
  // Once the state changes...
  state = reducer(state, action);

  // ...call the listeners!
  // Yes, it was that simple :).
  listeners.forEach(listener => listener());

  return action;
}
```

Notice how the listeners are called without arguments. This is because there is no need to do so. Essentially, whomever subscribed to the store changing also has access to getting its new state. With this lazy approach we save some extra complexity by not assuming what the callback wants.

This section was inspired in Dan Abramov's talk at ReactEurope 2016. We recommend you watch it as it explores the benefits of Redux constraints from an interesting perspective that could make you appreciate the library and its simplicity even more: **https://github.com/gaearon/the-redux-journey**.

Connecting Redux to our UI

Now that we have a way to understand how our app changes in a structured way, we need to display that somehow and interact with the user.

To get started, we will work with a very basic view engine that renders to `console.log`, and we will gradually transition to `React` and its official binding with `redux`: `react-redux`. Understanding how the mechanism works from scratch allows you to connect our data in a more sensible way going forward and detect potential bottlenecks.

The basic renderer

Continuing with our example above, we'll write a view function that shows characters as we type them. Let's define our basic renderer:

```
function View(text) {
  console.log(`"${text}"`);
}
```

This is nothing really fancy, just piping in the input text to the console and wrapping it with double quotes.

Our store will be a slightly simplified version of the editor example above:

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

const reducer = (state = '', action) => {
  switch (action.type) {
    case 'CHARACTER_TYPED':
      return state + action.char;

    case 'BACKSPACE':
      return state.substr(0, state.length - 1);

    default:
      return state;
  }
}

const store = createStore(reducer);
```

Now that we have a store and a view, let's do a first render:

```
// get the text
const text = store.getState();
// and render it!
View(text);
// the console should've logged an empty string
```

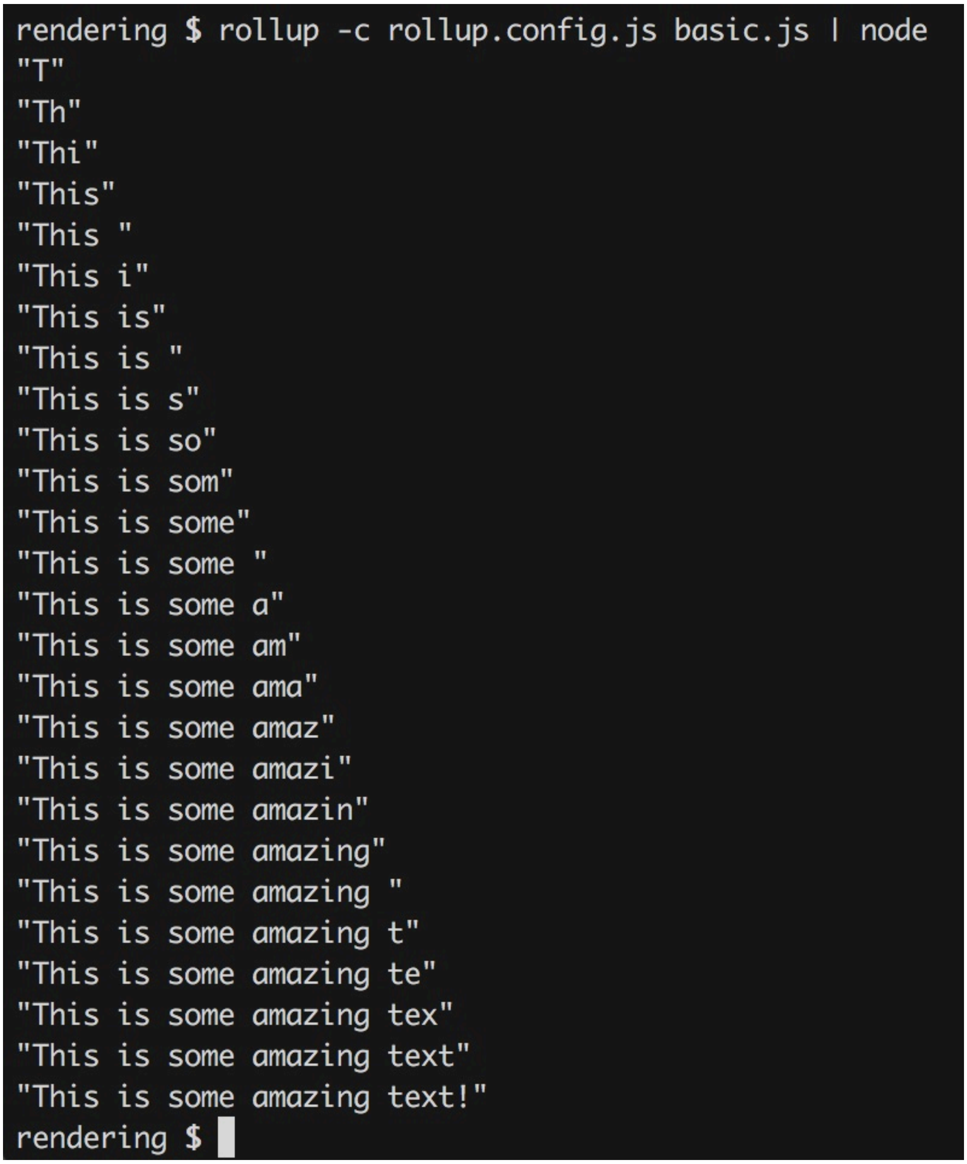
Let's simulate the user typing in some text through time by using the `insertCharacter` action we defined previously:

```
const userInput = `This is some amazing text!`;
let index = 0;
const interval = setInterval(() => {
  // if we still have text to process
  if (index < userInput.length) {
    // tell the reducer to add the character
    store.dispatch(insertCharacter(userInput[index++]));
  } else {
    // otherwise stop
    clearInterval(interval);
  }
  // do this every 0.25 seconds (or 250ms)
}, 250);
```

Our view is rather static, because it is in fact a function, and unless we call, it won't know that it should be showing anything different to the user. Luckily for us, whenever the state changes in Redux we get a chance to do something with that change through `store.subscribe`. Let's see how that works:

```
// let's abstract our rendering logic above into a rendering function that
gathers data for a view
// and renders it to the user
function render() {
  const text = store.getState();
  View(text);
}

// when the store's data changes we'll run our rendering function effective-
ly turning it into a
// rendering loop
store.subscribe(render);
```

Go ahead and try that yourself. Here's what the final result looks like:



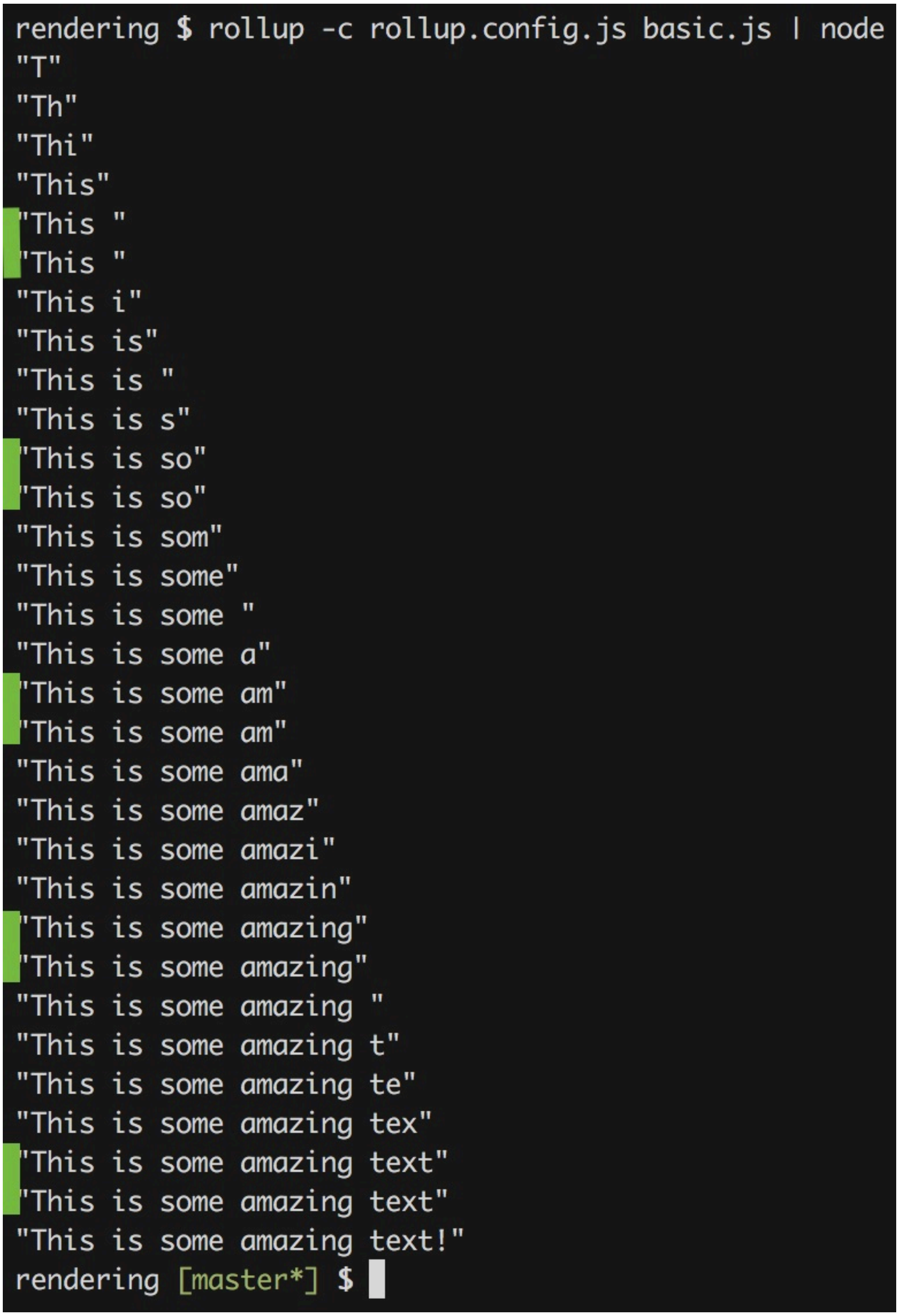
```
rendering $ rollup -c rollup.config.js basic.js | node
"T"
"Th"
"Thi"
"This"
"This "
"This i"
"This is"
"This is "
"This is s"
"This is so"
"This is som"
"This is some"
"This is some "
"This is some a"
"This is some am"
"This is some ama"
"This is some amaz"
"This is some amazi"
"This is some amazin"
"This is some amazing"
"This is some amazing "
"This is some amazing t"
"This is some amazing te"
"This is some amazing tex"
"This is some amazing text"
"This is some amazing text!"
rendering $
```

In a more complex application, a user would probably be doing more things that eventually dispatch other actions through our store. However, our rendering loop will always run, and despite the fact that our View's data hasn't changed, it will still run it. In real life we'll have many views present on screen, of which most of them will be somehow connected to the store and listening for data changes. So it's our responsibility to ensure that we don't waste the user's computational resources in operations that don't need to happen in the first place, and leave room for those who do.

Let's expand our fake user interaction from above to include some other type of actions being dispatched from time to time:

```
const userInput = `This is some amazing text!`;
let index = 0;
// our random tracker
let nextIsRandom = false;
const interval = setInterval(() => {
  // if we still have text to process
  if (index < userInput.length) {
    if (nextIsRandom) {
      // tell the reducer to add the character
      store.dispatch({
        type: 'RANDOM'
      });

      nextIsRandom = false;
    } else {
      // tell the reducer to add the character
      store.dispatch(insertCharacter(userInput[index++]));

      // flag every fifth character to be random
      nextIsRandom = index % 5 === 0;
    }
  } else {
    // otherwise stop
    clearInterval(interval);
  }
  // do this every 0.25 seconds (or 250ms)
}, 250);
```

Here's the output with the unnecessary re-renders flagged in green:



```
rendering $ rollup -c rollup.config.js basic.js | node
"T"
"Th"
"Thi"
"This"
"This "
"This "
"This i"
"This is"
"This is "
"This is s"
"This is so"
"This is so"
"This is som"
"This is some"
"This is some "
"This is some a"
"This is some am"
"This is some am"
"This is some ama"
"This is some amaz"
"This is some amazi"
"This is some amazin"
"This is some amazing"
"This is some amazing"
"This is some amazing "
"This is some amazing t"
"This is some amazing te"
"This is some amazing tex"
"This is some amazing text"
"This is some amazing text"
"This is some amazing text!"
rendering [master*] $
```

Hopefully that example, despite its simplicity, can illustrate the re-rendering problem. In order to render the bare minimum, we'll need to keep track of data changes to determine whether we should render or not. Our render function seems like a great candidate for it, since it already takes care of getting the data and showing it. Let's refactor it to meet our needs:

```
// keep track of the previous text, i.e., the text that was rendered on the
last run
let prevText;
function render() {
  // get the text
  const text = store.getState();

  // compare them!
  if (prevText !== text) {
    // if it changed, save the new text for later reference
    prevText = text;
    // and render the changes
    View(text);
  }
}
```

Our output should look exactly the same as it did before the introduction of an extra action being dispatched.

The path to React

Now that we know the basics of rendering our potential UI with Redux, let's get that editor into a more useful renderer like React. This time, instead of faking the user's input, we'll actually listen to what they type on the screen.

In our previous example, both the subscription to the Redux store and the reference to the previous text were part of our scope. React allows us to encapsulate that within a component that makes it easier to reuse, so let's do that instead. It's refactor time!

```
// we've extracted the store into its own file for this example
import { insertCharacter, removeCharacter, store } from './store';
import { render } from 'react-dom';
import React, { Component, PropTypes } from 'react';

class View extends Component {
  constructor(props, context) {
    super(props, context);

    // the store comes as a prop to our component
    const { store } = props;
    // get the initial state from the store
    this.state = {
      text: store.getState()
    };

    // subscribe to changes and set the component's state when anything
changes
    this.cancelSubscription = store.subscribe(() => {
      this.setState({
        text: store.getState()
      });
    });

    this.onCharacter = this.onCharacter.bind(this);

    // listen to keystrokes in the
    document.addEventListener('keyup', this.onCharacter);
  }

  componentWillUnmount() {
    // clean up our binding to keydown on the document
    document.removeEventListener('keyup', this.onCharacter);
    // also clean up our subscription to the store
    this.cancelSubscription();
  }

  onCharacter(event) {
    const { dispatch } = this.props.store;

    if (event.key === 'Backspace') {
      // if the user pressed the backspace, remove the last character
      dispatch(removeCharacter());
    } else if (event.key.length === 1) {
      // otherwise, when a keystroke came our way, add it!
      dispatch(insertCharacter(event.key));
    }
  }

  render() {
    const { text } = this.state;

    // the example includes styles that are omitted for brevity
    return (
      <div>
        {text}
      </div>
    );
  }
}
// tell React that we're expecting a redux store like prop called store
```

```
View.propTypes = {
  store: PropTypes.shape({
    dispatch: PropTypes.func.isRequired,
    getState: PropTypes.func.isRequired,
    subscribe: PropTypes.func.isRequired
  }).isRequired
};

// render our component
render(
  <View store={store} />,
  document.body
);
```

Our React component above abstracts the complexity of dealing with the Redux store. However, as our application grows, connecting to the store will become a repetitive task and, if repeated along across components, it will make our logic more complex and error prone. It is also a good practice to try and keep our React components as stateless as possible, and our component above could have been as simple as: `const View = ({ text }) => <div>{text}</div>`.

That makes it easier to test it, and to separate concerns as we can compose that component more easily.

There are two key parts to our component: 1) Knowing that we have a Redux store, connecting to that store and updating the view with its changes 2) Rendering the text on screen as the user types

Let’s separate those into their own components, shall we?

Our View needs to know that a new key was pressed and tells the store about it; it also needs to render the current text. That translates to roughly the following component:

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

class View extends Component {
  constructor(props, context) {
    super(props, context);
    this.onCharacter = this.onCharacter.bind(this);
    // listen to keystrokes in the
    document.addEventListener('keyup', this.onCharacter);
  }

  componentWillUnmount() {
    // clean up our binding to keydown on the document
    document.removeEventListener('keyup', this.onCharacter);
  }

  onCharacter(event) {
    const { props } = this;
```

```
      if (event.key === 'Backspace') {
        // if the user pressed the backspace, remove the last character
        props.removeCharacter();
      } else if (event.key.length === 1) {
        // otherwise, when a keystroke came our way, add it!
        props.insertCharacter(event.key);
      }
    }

    render() {
      const { text } = this.props;

      return (
        <div>
          {text}
        </div>
      );
    }
  }
  View.propTypes = {
    insertCharacter: PropTypes.func.isRequired,
    removeCharacter: PropTypes.func.isRequired,
    text: PropTypes.string.isRequired
  };
```

Notice how our View now doesn't take a `store` as props anymore, but instead it takes `text` and functions to insert and remove characters. Congratulations. As it doesn't know anything about the store, we've just made our component reusable!

Now it's time for our store connector. We know that we have to send the view a part of the state and some functions that will eventually dispatch an action in our store. Because we can't be certain as to what part of the state or actions our view will need, we should give it the freedom to define them. We will use functions to achieve that. As a rough first approximation, we can probably say that such connector's function signature could be along the lines of:

```
connect(ComponentToConnect: ReactComponent, mapState: Function, actionsToDis-
patch: Object) :
ReactComponent
```

Ideally, we would wrap our view in this connected component with something like this:

```
const ConnectedComponent = connect(
  // the component we want to connect
  View,
  // the piece of the state we want to get back
  state => ({
    text: state
  }),
  // the actions that we want to dispatch
  {
    insertCharacter,
    removeCharacter
  }
);
```

Now that we know how we would want to use it, let's try and implement our `connect` function.

```
function connect(ComponentToConnect, mapState, actionsToDispatch) {
  class ConnectedComponent extends Component {
    constructor(props, context) {
      super(props, context);

      // the store comes as a prop to our component
      const { store } = props;

      // get the initial state from the store
      this.state = store.getState();

      // subscribe to changes and set the component's state when anything
changes
      this.cancelSubscription = store.subscribe(() => {
        this.setState(store.getState());
      });

      // map the actions that the view wants to dispatch to the store
      this.actions = {};
      Object.keys(actionsToDispatch).forEach(action => {
        this.actions[action] = (...args) => {
          store.dispatch(
            actionsToDispatch[action](...args)
          )
        };
      });
    }

    componentWillUnmount() {
      // clean up our subscription to the store
      this.cancelSubscription();
    }
    render() {
      // our component should be invisible so we should proxy the props that
we get from above
      // and the sliced state that we captured from our store and filtered
as the view wanted
      return <ComponentToConnect {...this.props} {...this.state} />;
    }
  }

  // tell React that we're expecting a redux store like prop called store
  ConnectedComponent.propTypes = {
    store: PropTypes.shape({
      dispatch: PropTypes.func.isRequired,
      getState: PropTypes.func.isRequired,
      subscribe: PropTypes.func.isRequired
    }).isRequired
  };

  return ConnectedComponent;
}
```

We're almost set! Our components are now rendering, and they are isolated from our Redux store. There's one issue though, as we go deeper into our React tree, we would have to remember to pass down the `store` prop so that every other component can also connect to it. This quickly becomes very tedious and error prone because it's very easy to forget to pass the store every time. Needless to say, it makes your components' logic muddy right away.

There's one more refactor that we can make: we can use React's context to make the store only be available to views in the React tree at request. This allows us to forget about passing down the store prop, and it will define the scope for our connection with Redux very well as only the the connect function will need to know about it. Win-win! Let's see what this root store provider looks like:

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

class Provider extends Component {
  constructor(props, context) {
    super(props, context);
    // get the store from the props
    this.store = props.store;
  }

  // expose the store as a child context for components that request it
  getChildContext() {
    return {
      store: this.store
    };
  }

  // only take one child and render it
  render() {
    return Children.only(this.props.children);
  }
}
```

```
// define the props that we take as a component
Provider.propTypes = {
  store: PropTypes.shape({
    dispatch: PropTypes.func.isRequired,
    getState: PropTypes.func.isRequired,
    subscribe: PropTypes.func.isRequired
  }).isRequired,
  children: PropTypes.element.isRequired
};
// define the prop types that children can claim through context
Provider.childContextTypes = {
  store: PropTypes.shape({
    dispatch: PropTypes.func.isRequired,
    getState: PropTypes.func.isRequired,
    subscribe: PropTypes.func.isRequired
  }).isRequired
};
```

Our connector also has to be refactored so that it gets its store from the context, instead of its props.

```
function connect(ComponentToConnect, mapState, actionsToDispatch) {
  class ConnectedComponent extends Component {
    constructor(props, context) {
      super(props, context);

      // the store comes as a prop to our component
      const { store } = context;

      // ...omitted for brevity
    }

    // ...omitted for brevity
  }

  // notice how we declare contextTypes instead of propTypes here
  ConnectedComponent.contextTypes = {
    store: PropTypes.shape({
      dispatch: PropTypes.func.isRequired,
      getState: PropTypes.func.isRequired,
      subscribe: PropTypes.func.isRequired
    }).isRequired
  };

  return ConnectedComponent;
}
```

Now in our final render method, we wrap our component with the Provider passing the store as its prop:

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

Let’s recap what we’ve done so far. We started with a React component that was doing everything from getting new data from the store to rendering it. We gradually extracted the key pieces until we get to three main concepts: `Provider`, `connect` and our `View`. In short, `Provider` is our bridge to tell `connect` which store to use when a `View` uses it.

Luckily for us, the Redux community has already sorted this problem and packaged it into `react-redux`. The solution is quite similar to what we got to with a difference in how `connect` works, which instead of having a signature like the following:

```
connect(ComponentToConnect: ReactComponent, mapState: Function, actionsToDis-
patch: Object) :
ReactComponent
```

It instead has this:

```
connect(mapStateToProps: Function, mapDispatchToProps: Object | Function) :
Function
// the returned Function has a signature of
connected(ComponentToConnect: ReactComponent) : ReactComponent

// mapStateToProps has the following signature
mapStateToProps(state: Object, props: Object) : Object

// mapDispatchToProps is generally used in one of two ways
// by providing a function that gets a dispatch method and you're in charge
of mapping to an object,
// this is handy if you need to transform data before passing it into the
action creator before
// calling it
mapDispatchToProps(dispatch: Object, props: Object) : Object
// or it takes an object with actions that the library will auto-map to dis-
patch as we did in our
// example above
mapDispatchToProps = actions: Object
```

In order to use `react-redux`, the only change we need to make to our View is on the connected component:

```
const ConnectedComponent = connect(
  // the piece of the state we want to get back
  state => ({
    text: state
  }),
  // the actions that we want to dispatch
  {
    insertCharacter,
    removeCharacter
  }
)(View);
```

In practice, it is a good idea to extract `mapStateToProps` and `mapDispatchToProps` into their own functions so that they're easier to read. So the example above would look like this:

```
const mapStateToProps = state => ({
  text: state
});

const mapDispatchToProps = {
  insertCharacter,
  removeCharacter
};

const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)
(View);
```

This also allows us to easily test both functions in isolation:

```
test('mapStateToProps', ({ end, deepEquals }) => {
  deepEquals(
    mapStateToProps('TEXT'),
    { text: 'TEXT' }
  );

  end();
});
```

In future chapters we will dive into how to use state selectors within `mapStateToProps` to achieve a more idiomatic definition, and in many cases get performance gains while rendering.

The path to Angular 2

We're now going to look at how we can implement the previous example in an Angular 2 application. We're going to be using the Angular 2 CLI to create the project. Be sure to read the prerequisites for the Angular CLI, which you can find on the Angular CLI github page. Explanations about the CLI and Angular in detail are out of the scope for this book.

You can install the Angular 2 CLI with the following command.

```
npm install -g angular-cli
```

Once it's installed we're ready to begin.

CREATING THE PROJECT

Let's begin by creating a new Angular 2 project. You can create a new project using the command:

```
ng new redux-ng2-example
```

The project will be created inside a new folder called `redux-ng2-example` relative to the current working directory. Once created, change your current working directory to the new project directory.

Test your app is working by running the following command:

```
ng serve
```

You should see a simple "app works!" message on **http://localhost:4200/**.

Note: At the time of writing when you change directory into `redux-ng2-example` you might also need to run:

```
npm install
```

This will install the missing project dependencies. This is a known issue that should hopefully be resolved soon. You may not need to do this step at the time of reading. If the application launches without any errors, you're okay.

CREATE A TEXTCOMPONENT

We need a text component to render the output from our previous example. We can generate a component using Angular CLI with the following command:

```
ng generate component text
```

This creates a new component called `TextComponent`. You can see in your terminal this component in a folder at `src/apps/text`.

ADDING THE TEXTCOMPONENT TO THE APP COMPONENT

Let's wire this component up so it renders in the UI.

Open `src/app/app.component.ts` and amend the file with the following changes:

```
import { Component } from '@angular/core';
import { TextComponent } from './text';

@Component({
  // ... unchanged properties omitted ...
  directives: [TextComponent]
})
export class AppComponent {
  title = 'app works!';
}
```

We've made two changes to the original file.

1. We imported the `TextComponent` module.
2. We added a new `directives` array to the `@Component` decorator.

The `directives` array informs Angular what components this component uses, and will allow us to use the `<app-text>` element in it's HTML template. Now that the app component is configured to use the text component, you can update the app component template and render the text component. Open `src/app/app.component.html` and replace the existing content with the following:

```
<app-text></app-text>
```

Launching the app with `ng serve` should now display:

```
text works!
```

REMOVING THE APPCOMPONENT TITLE PROPERTY

When we replaced the contents of `src/app/app.component.html` we removed the `{{title}}` template expression. We're no longer using the `title` property, so we should remove it from the app component. Open `src/app/app.component.ts` and remove the `title` property from the `AppComponent` class, leaving the following empty class:

```
export class AppComponent {}
```

FIXING THE BROKEN TEST

If you were to run `ng serve` now it would fail. This is because we've broken one of the `AppComponent` unit tests by removing the `title` property. Let's fix this. Open up `src/app/app.component.spec.ts` and remove the following test:

```
  it('should have as title "app works!"',
      inject([AppComponent], (app: AppComponent) => {
    expect(app.title).toEqual('app works!');
  }));
```

If we run `ng serve` again, the tests should now pass and the application should launch.

MAKING THE TEXTCOMPONENT DYNAMIC

Now let's start preparing the text component for use. Let's begin by opening the component template and replacing the static text with a template expression. Open up `src/text/text.component.html` and replace the contents of the file with the following:

```
<p>
  {{text}}
</p>
```

This wasn't a big change, we simply removed the static text and replaced it with a template expression that evaluates to the value of the `text` property. We still need to define the `text` property. Open up `src/app/text/text.component.ts` and add the `text` property to the `TextComponent` class:

```
export class TextComponent implements OnInit {
  text = '';

  constructor() {}

  ngOnInit() {
  }

}
```

If we run the application again we'll see an empty UI now. This is fine, because it's the expected behavior since we've said that `text` is an empty string. You can verify this by checking the console to make sure there are no errors. If you're still not convinced, set a custom string as the default value for the `text` property and this will be rendered to the UI.

INSTALLING REDUX

We're now in a good position to start adding Redux to our application. Let's start by installing it. Type the following command into your terminal:

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

Redux is now installed, but we still need to configure SystemJS and the Angular build so we can use it. Open up `angular-cli-build.js` and add the following entry to the end of the `vendorNpmFiles` array:

```
'redux/dist/redux.js'
```

This tells the Angular build tool that we want to include Redux as a vendor package. Next, we need to tell SystemJS where to find Redux and what type of file to expect. Open up `src/system-config.ts` and add the following entry to the empty map object so that it looks like this:

```
const map: any = {
  'redux': 'vendor/redux/dist/redux.js'
};
```

This tells SystemJS where to look when the user imports the `redux` package. Next we need to tell SystemJS what type of module system Redux uses. Inside the same file, add the following entry to the empty packages object so it looks like this:

```
const packages: any = {
  'redux': {
    format: 'cjs'
  }
};
```

This tells SystemJS that Redux uses the CommonJS module format. We've now told Angular about our new vendor package, so this will get copied over when we build the project, and we've told SystemJS where to find Redux and what module format to expect when it finds it.

Running the server will verify nothing has broken so far. Remember to edit the text components string value to see something in the DOM.

CREATING THE STORE SERVICE

Redux is installed, configured and ready to use. We're now going to create a new service to manage the Redux store. Type the following command into your terminal:

```
ng generate service store
```

This creates a new service, but you still need to configure the application to use it. Open up `src/app/app.component.ts` and amend the contents to match the following:

```
import { Component } from '@angular/core';
import { TextComponent } from './text';
import { StoreService } from './store.service';

@Component({
  // ... default properties omitted  ...
  directives: [TextComponent],
  providers: [StoreService]
})
export class AppComponent {}
```

We've made two changes here, we added:

1. A new `import` statement to import the `StoreService` class.
2. The new `providers` array to the `@Component` decorator.

The providers array allows the `StoreService` to be injected into all of the application components.

INJECTING THE STORESERVICE

Now that we've added the `StoreService` to the app component providers array, we can inject the service in our text component. Let's do that now. Open up `src/app/text.component.ts` and make the following changes:

```
import { Component, OnInit } from '@angular/core';
import { StoreService } from './../store.service';

export class TextComponent implements OnInit {
  text = '';

  constructor(private storeService: StoreService) {}

  ngOnInit() {
  }

}
```

We've made two changes to this file:

1. We added a new `import` statement to import the `StoreService` class.

2. We updated the `constructor` function signature to inject the `StoreService` as a new private property called `storeService`.

At this point the `TextComponent` is ready to start working with Redux. We have access to the Redux store through the `StoreService` method, which gives us access to the state and allows us to dispatch actions and subscribe to the store.

FIXING THE TEXTCOMPONENT TEST

The previous change broke one of the `TextComponent` tests. This happened because we introduced a new required argument to the constructor. A failing test will stop the application from launching, so we need to fix this right away. Open `src/app/text/text.component.spec.ts` and and make the following changes:

1. Import the `StoreService` like we did in the text component.
2. Update the test to pass a new instance of the `StoreService` as the first argument of the `TextComponent` constructor.

The resulting code should look like this:

```
// ... omitted code we've not changed ...

import { StoreService } from './../store.service';

describe('Component: Text', () => {
  it('should create an instance', () => {
    let component = new TextComponent(new StoreService());
    expect(component).toBeTruthy();
  });
```

This fixes the broken test and allows us to build and run the application again.

ADDING BEHAVIOR TO THE STORE SERVICE

We're now going to extend our store service and add some real logic. Open `src/app/store.service.ts` and paste the following into the file:

```
import { Injectable } from '@angular/core';
import { createStore } from 'redux';

@Injectable()
export class StoreService {
  private store;
  constructor() {
    this.store = createStore(this.getReducer());
  }
```

```
    getReducer() {
          return (state = '', action) => {
        switch (action.type) {
          case 'CHARACTER_TYPED':
            return state + action.character;
          case 'BACKSPACE':
            return state.slice(0, -1);
          default:
            return state;
        }
      };
    }
    getStore() {
        return this.store;
    }
  }
```

There's five notable changes here:

1. We added a new import statement to import the `createStore` function from Redux.
2. We added a new private `store` property, which we'll use to hold a reference to the store instance.
3. We added a new method called `getReducer`. This method returns the root reducer for the application. This is the same reducer as the one used in the previous example.
4. We added a new method called `getStore`. This method returns the store instance. We'll be using this to access the store in the text component.
5. The constructor now creates a new store, using the reducer retuned from `getReducer`, and stores a reference to the new store in the `store` property.

ADDING ACTION CREATORS

Our component is going to dispatch actions to change the store, so let's create a new module to hold all of our action creators. Create a new file called `src/app/action-creators.ts` and add the following:

```
'use strict';

export const removeCharacter = () => ({
    type: 'BACKSPACE'
});

export const insertCharacter = (character: string) => ({
    type: 'CHARACTER_TYPED',
    character
});
```

ADDING BEHAVIOR TO THE TEXTCOMPONENT

We're now (finally) ready to hook our component up to Redux. We're going to update the `TextComponent` to capture keypresses and insert a character, and allow backspace to remove one.

Open `src/app/text/text.component.ts` and add the following:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { StoreService } from './../store.service';
import { removeCharacter, insertCharacter } from './../action-creators';

@Component({
  moduleId: module.id,
  selector: 'app-text',
  templateUrl: 'text.component.html',
  styleUrls: ['text.component.css']
})
export class TextComponent implements OnInit, OnDestroy {
  text = '';
  store = null;
  cancelSubscription = null;

  constructor(private storeService: StoreService) {}

  ngOnInit() {
      this.onCharacter = this.onCharacter.bind(this);
      this.store = this.storeService.getStore();
      this.cancelSubscription = this.store.subscribe(() => this.text =
this.store.getState());
      document.addEventListener('keydown', this.onCharacter);
  }
  ngOnDestroy() {
      this.cancelSubscription();
      document.removeEventListener('keydown', this.onCharacter);
  }
  getKey(event) {
      const isSpace = event.code === 'Space';
      if (isSpace) {
          return ' ';
      }
      const isAlphaKey = !!event.code.match(/Key([A-Z])/);
      if (isAlphaKey) {
          const key = String.fromCharCode(event.keyCode);
          return event.shiftKey ? key.toUpperCase() : key.toLowerCase();
      }
  }
  onCharacter(event) {
      if (event.code === 'Backspace') {
          // if the user pressed the backspace, remove the last character
```

```
            this.store.dispatch(removeCharacter());
        } else {
            const key = this.getKey(event);
            if (key) {
                this.store.dispatch(insertCharacter(key));
            }
        }
    }

}
```

That's a lot of code changes, let's break it down and discuss each change.

New Module Imported

We've added a new import statement to import the `removeCharacter` and `insertCharacter` action creators from the new `src/app/action-creators.ts` module. We've also extended the `@angular/core` import statement to import the `OnDestroy` interface.

New Class properties

We've added two new class properties called `store` and `cancelSubscription`. `store` is simply a cache/container for the Redux store so we don't have to keep using the service to get access to the store instance, and `cancelSubscription` just holds a reference to the store subscription so that we can call it during the `ngOnDestroy` lifecycle hook to unsubscribe from the store.

The ngOnInit method

We store a reference to the store in a property called `store`. This is for convenience only. We then subscribe to store changes with the `store.subscribe` method, which allows us to keep the component's `text` property in sync with the store. We store the result of calling `subscribe` (the subscription) in a property so that we can call it later. Finally we listen to the `keyup` event on the document, passing the `onCharacter` method as the callback handler for this event.

The ngOnDestroy method

We unsubscribe from the `store` by calling the stored subscription in `cancelSubscription`, and we unbind the `key` event listener from the document.

The getKey method

This is a helper method that receives a DOM event and returns a string representing the character that was pressed. This allows us to simplify the `onCharacter` method.

The onCharacter method

This is the event listener we added for the document `keyup` event. It dispatches one of two actions depending on the event code.

RUNNING THE FINISHED APPLICATION

That's it for this example. Try running the app with `ng serve`. You should be able to type on the keyboard and see the characters printed in the UI. Hitting backspace should delete characters too.

OPEN SOURCE SOLUTIONS

This section walked you through creating your own Redux powered Angular 2 application. This wasn't the most ideal implementation as we had to deal with boilerplate in both the TextComponent and StoreService.

The TextComponent in particular has some unnecessary boilerplate to manage both the store subscription and the store object. Using a library can help to reduce or remove the boilerplate completely.

There are a few options to help abstract away the boilerplate from our code. Let's take a quick look at two of the more popular ones to give you an idea of what's available.

Ng2 Redux (**https://github.com/angular-redux/ng2-redux**)

Ng2-redux provides Angular 2 bindings for Redux. This provides similar functionality to react-redux but for Angular 2. There are two patterns available for connecting your Angular 2 components to Redux, the connect pattern and the select pattern.

The connect pattern is similar to the connect pattern from react-redux, so this should be familiar to developers who have used those bindings before. However the recommended pattern is the select pattern, which has been created to provide the best performance with Angular 2's view layer.

NgRx store (**https://github.com/ngrx/store**)

While this is not a Redux bindings library, it's worth considering this if you're using Angular 2. NgRx Store is an RxJS powered Redux inspired state container for Angular 2 applications. If you're using RxJS extensively in your applications, you'll certainly want to check this one out.

COMPARING SOLUTIONS

Let's compare the solution we developed with one that uses `ng2-redux`. With `ng2-redux` we can eliminate a lot of boilerplate from the Text Component. We can also eliminate the need for a Store Service because the store will be managed by `ng2-redux`, so it will now take care of ensuring the store is accessible within our components.

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { removeCharacter, insertCharacter } from './../action-creators';
import { select, NgRedux } from 'ng2-redux';

@Component({
  // ... omitted unchanged code ...
})
export class TextComponent implements OnInit, OnDestroy {
  @select() text;

  constructor(private ngRedux: NgRedux<string>) {}

  ngOnInit() {
      this.onCharacter = this.onCharacter.bind(this)
      document.addEventListener('keyup', this.onCharacter);
  }
  ngOnDestroy() {
      document.removeEventListener('keyup', this.onCharacter);
  }
  private getKey(event) {
      // ... omitted unchanged code ...
  }
  onCharacter(event) {
      if (event.code === 'Backspace') {
          // if the user pressed the backspace, remove the last character
          this.ngRedux.dispatch(removeCharacter());
      } else {
          const key = this.getKey(event);
          if (key) {
              this.ngRedux.dispatch(insertCharacter(key));
          }
      }
  }

}
```

To be fair, there's still a lot of code here, but the majority of that code is business logic. There is much less Redux specific boilerplate. Here are the key differences between the two approaches:

1. There's no longer a store property on the TextComponent class.
2. We no longer need to subscribe to the store.
3. We no longer need to unsubscribe from the store.

All of these points relate to managing the store. Subscribing to and unsubscribing from the store was the main problem with managing the store ourselves, yet `ng2-redux` now takes care of this for us.

We do now have an `ngRedux` property, which has replaced our `StoreService` and provides a gateway to the Redux specifics, such as dispatch. If you look at the logic in the `onCharacter` method you'll see that we're now using `this.ngRedux.dispatch` instead of `this.store.dispatch` to dispatch an action to the store. We've actually used dispatch different to the `ng2-redux` recommended approach, but what we have is sufficient for this example.

Did you notice the `@select` decorator? This decorator connects the `text` property to the store. Anytime the `text` property on the store changes this `text` property will change too. This decorator turns your properties into observables.

The real question is, was this change to `ng2-redux` worth it? Considering that you'll have many components in a real application, and each of those components need to connect to a store, then yes, it is worth it. Whichever approach you take, you'll want to abstract the boilerplate out of your components to avoid repeating yourself. `ng2-redux` has already gone through the trouble of doing this for you, so why bother yourself?

State

In our above example, the state shape is extremely simple. However, as your app grows, the state shape your reducers will produce will get a lot more complex, and at that point it is worth thinking about the state shape a bit more strategically.

This is because even though reducers *produce* the state, the resulting state tree will directly affect how the remainder of the Redux-y parts of your app and will look like selectors, actions creators, middleware, etc.

Unsurprisingly, this means the state shape has a huge impact on how easy and pleasant it is to write your app as a whole. A state shape that has been thoughtfully designed to your app's requirements will make it fun and engaging; it will be the backbone of your application that you can always rely on. A state that has been haphazardly glued together from different concepts and ideas, however, might make it a convoluted nightmare that no one will want to touch.

Ok, so what makes a healthy state shape?

First and foremost, the two most common operations you will perform on it should be accommodated: transformation and consumption.

Transformation, as we saw previously in this chapter, is achieved through reducers. Lots of data — and lots of different *kinds* of data — will eventually co-exist inside the state: entities, UI state, session information, etc. Your state shape, and the corresponding reducers, should make it a breeze to transform any part of it without having to jump through hoops.

Consumption is done through selectors that expose state to your components. Your state shape should make it possible to create selectors that are easy to use and understand, and which don't tank your app's performance.

The catch here is that a state shape that is easy consume is not so easy to transform, and vice versa.

This is caused by the fact that transformations are best done on *normalized* structures (no duplication, no nesting), whereas consumption usually requires *nested* data (that is, after all, what your components need to render). This means that you either keep hierarchies inside your state and complicate your reducers, or you normalize your state, but increase complexity on the selector-side.

In general, it is recommended to lean towards an easy-to-transform state and more complex selectors, but we think getting stuck inside the ‘normalize everything’ mindset just because of a couple of StackOverflow answers point in that direction is not useful.

We don’t believe there is no *One True Way* to structure your state in Redux, just as there is no *One True Way* to compose your React components. Every app is different and so is its state. It will be a balancing act, where — in the end — you yourself will have to figure out what works best in your context.

Apart from transformation and consumption, there are also other considerations you might want to make for your state shape. Their importance will vary from app to app, so don’t feel like you have to tick all of the check-boxes.

Serialization. One very cool feature of Redux state is that it gives you complete freedom about what it can be — you can use a single string as we have seen in the beginning of this chapter, but most often you will find it will be a plain JavaScript object. This has some pretty cool implications. For instance, you can save the entire state to local storage and reuse it across sessions easily. You might also want to sent it along with your crash reports to superpower your remote debugging. If you want to achieve either of the above, the only thing to keep in mind is to keep your state serializable. Basically this means: dont’t store functions inside the state.

Performance. There are libraries, most notably Immutable.js, that can improve the performance of your state transformations. Nevertheless, to achieve this, they will wrap your state with an immutable container. The tradeoff here is that things like logging, or the above mentioned serialization, become somewhat more challenging because you are not just dealing with simple JavaScript objects.

Lastly, I think it is helpful to think about your state as a fluid thing. Keep it appropriate for your current implementation. It is easy to fall into the trap of overdesigning it at the start. Just keep it lean and make pragmatic decisions as your app grows.

Conclusion

You now have a good understanding of the core concepts in Redux, including actions, reducers, stores and middleware. In the next chapter we detail Redux patterns, which will help reduce some of the code that you use in Redux.

Redux Patterns 3

There are a number of patterns worth learning about that are commonly found in Redux applications. These patterns are usually about reducing the amount of code you need to write to perform small, common tasks. Usually these patterns require the help of ES6 to simplify and reduce the amount of code you write. In fact, it's common to see ES6 heavily used in Redux, so you need to be comfortable reading it.

Note: You can find the complete source code for the sample application at **https://github.com/arturmuller/developing-a-redux-edge**.

It's important to really understand these patterns because it will make reading and writing Redux application code much easier. Before we look at them, let's quickly review the ES6 features that are commonly used to create the patterns:

- Object Destructuring
- Object Spread Operator (in-proposal)
- Array Spread Operator
- Arrow Functions
- Computed Property Names

This isn't an ES6 primer, so we won't be covering these features in this book. There are, however, plenty of resources available online where you can learn more about them. Instead we're going to take a look at how these features are commonly used in Redux.

Tip: Use Babel's online parser to test these examples out. Make sure you understand what's going on behind the scenes. You can try this out for yourself at **https://babeljs.io/repl/**.

Improving functions

The following patterns are used to improve the functions that you write. Improvements here usually cover reducing the size of functions. These patterns reduce the amount of code you need to write, while also improving the readability of the functions.

Concise functions

Redux embraces functional programming, which results in more functions that are smaller. Given that you'll be writing more functions, it's a good idea to use the best syntax for the job. In Redux, it's very common to see arrow functions used to make functions more succinct. Consider the following action creator that has been written without the use of arrow functions:

```
export const removeNote = function (id) {
  return {
    type: 'app/removeNote',
    payload: { id },
  };
}
```

It's not bad, but it can be better. Considering how often you'll write functions like the one above, you want to use the most concise syntax available. Arrow functions allow you to simplify the above example, thus reducing the amount of code that you write. Compare the previous action creator with the following one, which uses arrow functions:

```
export const removeNote = (id) => {
  return {
    type: 'app/removeNote',
    payload: { id },
  };
}
```

It's somewhat better, but at the moment all we've really accomplished is the removal of the function keyword. It's common to take this a step further. If an arrow function only consists of a return statement, you can omit the return statement and make it implicit:

```
export const removeNote = (id) => ({
  type: 'app/removeNote',
  payload: { id },
})
```

This is better. In this example we've completely removed the function keyword and return statement. Notice that we've also wrapped the curly braces in parenthesis too. We had to do this in order to prevent it from being parsed as a block statement. You'll see this pattern used a lot in Redux, so make sure that you understand what's going on here.

Extracting parameters

Another type of function you will be writing is a reducer function. These functions are passed an action object as their second argument. This action object is used as a container, and you usually extract a number of properties from it.

Rather than manually extracting object properties into local variables, you can instead use object destructuring. With this pattern you can extract properties in the function signature rather than adding (unnecessary) additional lines of code to your reducers.

For example, consider the following reducer that extracts properties from the action object into local variables:

```
export const addTodo = (state = [], action) => {
  const type = action.type;
  const todo = action.todo;
  switch (type) {
    case 'addTodo':
      return [...state, todo];
    default:
      return state;
  }
};
```

This isn't terrible, but we've added boilerplate to the reducer, which is reducing the readability of the reducer. Instead, compare the previous example to the following one, which uses object destructuring to extract these properties:

```
export const addTodo = (state = [], { type, todo }) => {
  switch (type) {
    case 'addTodo':
      return [...state, todo];
    default:
      return state;
  }
};
```

This is a big improvement. We've removed 2 lines of code from our reducer, which reduces noise and results in a much cleaner function. This is a very common pattern and it's not unusual to see all reducers follow this approach.

Reducer patterns

Reducers often perform a simple task, such as adding an element to an array or removing a property from an object. At the same time, reducers should take care not to mutate existing state, meaning that they always return new objects or arrays if they need to change them. This fact usually complicates the logic and makes the usual approaches obsolete. The following patterns are commonly found in reducers to help achieve these goals.

Adding elements to arrays

It's very common to add elements to an array in your reducers. The following pattern is typically used for this purpose.

```
export const addTodo = (state = [], { type, todo }) => {
  switch (type) {
    case 'addTodo':
      return [...state, todo];
    default:
      return state;
  }
};
```

The above code uses the array spread operator to add an element to an array. To achieve this without the array spread operator would require creating a duplicate of the array and manually pushing the new element into the array:

```
    export const addTodo = (state = [], { type, todo }) => {
      switch (type) {
        case 'addTodo':
          const newState = state.slice();
          newState.push(todo);
          return newState;
        default:
          return state;
      }
};
```

As you can see, the array spread operator reduces the amount of code that we need to write when adding new elements to arrays while avoiding mutations.

Removing elements from arrays

Another common pattern is to remove elements from arrays. There are a few approaches to this problem depending on your use case.

To remove an element from the end of an array you can use the `Array.slice` method:

```
const removeLastItem = (items) => {
    return items.slice(0, -1);
}
```

To remove an element from the front of the array you can use the ES6 `array spread` feature:

```
const removeFirstItem = (items) => {
    const [last, ...rest] = items;
    return rest;
}
```

To remove an element by a predicate you can use `Array.filter`:

```
const removeItemById = (items, id) => items.filter(item => item.id !== id);
```

Changing object properties

The following pattern is typically used in reducers when updating object properties. It allows you to easily change the property of an object while enforcing immutability.

```
const toggleTodo = (todo) => ({
    ...todo,
    completed: !todo.completed
  });
```

The above code creates a new object that consists of every property from the existing `todo` object. We then overwrite the `completed` property with the value of the `todo.completed` property inverted. Notice that we're also using the arrow function pattern mentioned previously.

This pattern also uses the object spread operator, which is currently in- proposal. If you prefer to avoid using the object spread operator, you can use `Object.assign` as an alternative:

```
const toggleTodo = (todo) => (Object.assign({}, todo, {
  completed: !todo.completed
}));
```

Note:`Object.assign` is part of ES6, but browser support may vary. You may need to include a polyfill if you chose to use `Object.assign`.

Adding properties to objects

The same pattern can also be used to add new properties to objects. For example:

```
const addTodo = (todos, todo) => ({
  ...todos,
  [todo.id]: todo
});
```

The above code adds a new `todo` to the `todos` object, using the id of the todo as the property name. You can also use the `Object.assign` alternative too if you prefer:

```
const addTodo = (todos, todo) => (Object.assign({}, todos, {
  [todo.id]: todo
}));
```

The only difference between this and the changing properties on the objects example is that this one is using a computed property name. We only needed to do that here because the `todos` object is being used as a map.

Removing properties from objects

When an object is used as a map, which is common, there is often logic required to remove properties from the object. This is usually easy to do, but the rules of immutability make this a little more difficult in Redux. The following pattern can be used for this purpose, because it's a concise solution for removing a property from an object without mutating the original object.

```
export const removeTodo = (state = {}, { type, id }) => {
  switch (type) {
    case 'removeTodo':
      const {[id]: remove, ...rest} = state;
      return rest;
    default:
      return state;
  }
};
```

It might be difficult to see what's going on here at first, because we're using a combination of object spread, object destructuring, and computed properties.

Basically, we're extracting a property from the `state` object by name (using a computed property to provide the name) and storing that value in the `remove` variable. At the same time, we're using the object rest to extract the remaining properties (every other property except the one extracted with `remove`) and storing those as a new object in the `rest` variable.

Finally, we're returning the `rest` variable as the new state object. In the end, the `rest` variable represents the collection with the specified item removed.

You can achieve the same thing without using ES6 features, but it's not as concise. We have to manually create a new object, copy the properties over and then delete the properties we no longer require:

```
export const removeTodo = (state = {}, { type, id }) => {
  switch (type) {
    case 'removeTodo':
      const newState = Object.assign({}, state);
      delete newState[id];
      return newState;
    default:
      return state;
  }
};
```

Conclusion

We've covered many common patterns that you'll likely run into as you learn and read more about Redux, and the majority of them use ES6 features to help keep the code concise. It is a lot of information to digest, but it's worth spending the time to familiarize yourself with these patterns and where they are used. In the next chapter we will create our example app, which will put to use the concepts covered thus far.

Building the Sample App 4

Having learned about Redux core concepts and how to connect our UI to the store, it is time to build an actual application to deepen our knowledge and understanding of the previous chapters. Follow along as we walk you through the steps required to build the example app.

Note: You can find the complete source code for the sample application at **https://github.com/arturmuller/developing-a-redux-edge**.

Application specification

Our application is going to be a small notes app, and our minimum viable product will — as the name suggests — only contain the bare minimum of functionality for now.

Actions

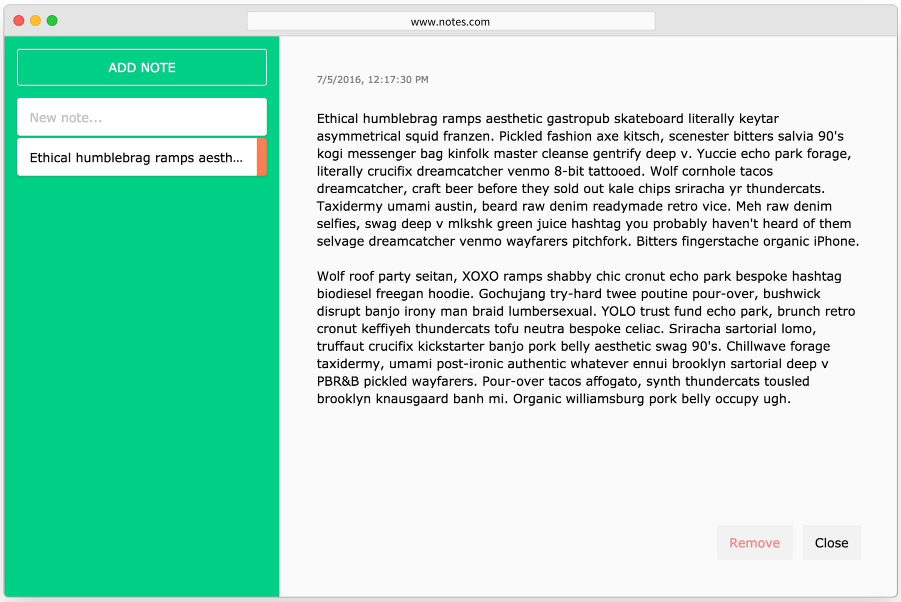
In order to create user interaction for our app, we need some actions that we can trigger in order to update our application state.

In our MVP these actions will be:

- Adding a new, empty note to the store.
- Updating an existing note in the store with new content.
- Removing a note from the store.
- Displaying a note in the details view.
- Clearing the detail view.

Views

The last two actions have already hinted at the fact that we also need some views for our application. But you may have guessed that too, because what good is an application if there is no UI to interact with it? Our application will use a simple master-detail layout:



7/5/2016, 12:17:30 PM

Ethical humblebrag ramps aesthetic gastropub skateboard literally keytar asymmetrical squid franzen. Pickled fashion axe kitsch, scenester bitters salvia 90's kogi messenger bag kinfolk master cleanse gentrify deep v. Yuccie echo park forage, literally crucifix dreamcatcher venmo 8-bit tattooed. Wolf cornhole tacos dreamcatcher, craft beer before they sold out kale chips sriracha yr thundercats. Taxidermy umami austin, beard raw denim readymade retro vice. Meh raw denim selfies, swag deep v mlkshk green juice hashtag you probably haven't heard of them selvage dreamcatcher venmo wayfarers pitchfork. Bitters fingerstache organic iPhone.

Wolf roof party seitan, XOXO ramps shabby chic cronut echo park bespoke hashtag biodiesel freegan hoodie. Gochujang try-hard twee poutine pour-over, bushwick disrupt banjo irony man braid lumbersexual. YOLO trust fund echo park, brunch retro cronut keffiyeh thundercats tofu neutra bespoke celiac. Sriracha sartorial lomo, truffaut crucifix kickstarter banjo pork belly aesthetic swag 90's. Chillwave forage taxidermy, umami post-ironic authentic whatever ennui brooklyn sartorial deep v PBR&B pickled wayfarers. Pour-over tacos affogato, synth thundercats tousled brooklyn knausgaard banh mi. Organic williamsburg pork belly occupy ugh.

NOTES LIST

The notes list is located on the left side of the screen. It shows a list of all notes that are residing in our store. In addition to that it also contains an “Add note” button, which allows you to create new notes.

NOTE DETAIL

The detail view shows the content of a note and also doubles as an editor where you can modify the note’s contents. The “Remove” and “Close” buttons do exactly what it says on the tin. The former removes the note from our store and the latter clears the note from being displayed in the detail view.

Storage

For now we will store all of our state in memory on the client. Later on we will be looking into storing our notes on a remote server, but for the MVP we will keep things simple.

Setting up

Now that we have clarified our brief, let's also go over our stack and methodology.

Methodology

We will be using Test Driven Development (TDD) to make sure our application is accurate, well modularized, and maintainable.

TDD is a popular software development methodology in which you write tests *before* you implement features. This has several benefits:

1. Writing tests at the start essentially makes you create a specification of your desired features before you actually write the implementation. Once you actually write your application code, you will have a clear idea of what you want to build, which makes building it that much faster and fun.
2. Because it is easier to write several small tests, instead of a single complex one, tests steer you toward small, focused modules. This prevents tight coupling, and smaller, composable modules are also easier to reason about.
3. Tests serve as a living documentation of your application. This is a great benefit for maintainability. Once you start refactoring parts of your app, it is great to have a reference point with what exactly the intended purpose is of the given module, instead of having to guess purely based on the implementation.

Writing tests in Redux is really easy. Given how most of the time you work with pure functions, assertions become simple comparisons between *actual* and *expected* outputs of the tested module. It really doesn't get much simpler than that.

Stack

In addition to Redux, we will use several other libraries to build our app.

BABEL

It is entirely possible to write Redux apps in ES3/5 JavaScript, but ES6 (aka ES2015) introduces several handy additions to the language that make combining it with Redux particularly useful.

Babel is a JavaScript *transpiler*, which means it 'translates' our future JavaScript code to something any browser can understand.

For our application code, we will use Babel through a webpack's `babel-loader` (see more below), and for our tests, we will use `babel-register` to transpile our code on the fly.

Babel allows for extensive configuration, but we are sticking to the basics: we want to use any feature that is a part of the ES2015 standard, React's JSX syntax, and the experimental object-spread syntax. This setup can be found inside the `.babelrc` file in the root of our app directory.

Learn more about Babel here: **https://babeljs.io/**.

WEBPACK

Webpack will be used to bundle our entire codebase into a single `app.js` file, which can then be easily referenced from a `<script>` tag inside our static HTML index file.

Webpack is an incredibly powerful tool, but it can be easy to get lost in the options it offers. For the purposes of this book, the setup has been kept to a minimum:

```
const path = require('path');

// The `entry` option specifies where webpack should start crawling
// our dependency graph
const entry = [ './source/index' ];

// The `output` option specifies where the final bundled js file will be
// placed, and how it will be called.
const output = {
  path: path.resolve(__dirname, 'public/js'),
  filename: 'app.js',
};

// The `resolve` option let's webpack know which file extensions should it
// be looking at. Note that the empty string has to be included, otherwise
// referencing external libraries (Eg: `import React form 'react'`) would
// not work.
const resolve = {
  extensions: [ '', '.js', '.jsx' ],
};

// webpack has a concept of loaders, which allow you to preprocess
// files before webpack bundles them. By default, webpack only works with
// ES5, so we will use Babel to transpile our code before letting webpack
// do its thing.
const scriptLoader = {
  loader: 'babel',
  include: path.resolve(__dirname, 'source'),
  test: /\.jsx$|\.js$/,
};

// The `devtool` option helps us debug our code by enabling various
// ways in which we can map the original source code to the bundled file.
// 'eval' is the simplest and fastest option which, in case of an error,
// will simply point you towards the module in which the error occurred.
const devtool = 'eval';
```

```
// Finally, we export the entire config so that webpack can actually
// use it!
module.exports = {
  entry,
  output,
  resolve,
  module: { loaders: [ scriptLoader ] },
  devtool,
};
```

Find out more about Webpack here: **https://webpack.github.io/**.

TAPE

Tape is a versatile, simple, and transparent testing library. It doesn't rely on magical globals like `describe` or `it`, but it comes with its own assertions.

Just like any other library in your codebase, it is used by simply importing it. Absolutely zero hassle here. Tape outputs the tried and true TAP (Test Anything Protocol), which — even on its own — is very human-readable. If you want to sweeten to deal though, there is a whole legion of tools that consume TAP output and transform it in some way. Check out `faucet` or `tap-pessimist`.

An example tape test looks like so:

```
// getFoo.test.js

import test from 'tape';
import getFoo from './getFoo';

test('getFoo', ({ equal, end }) => {
  const actual = getFoo();
  const expected = "foo";

  equal(actual, expected);
  end();
 });
```

To run the test, fire off the following in your terminal (assuming you have `tape` and `babel-register` installed):

```
$ tape --require babel-register getFoo.test.js
```

You will get the following output:

```
TAP version 13
# getFoo
ok 1 should be equivalent
```

```
1..1
# tests 1
# pass  1

# ok
```

If `getFoo()` incorrectly returns `"bar"` instead of `"foo"`, you will get the following error report:

```
TAP version 13
# getFoo
not ok 1 should be equal
  ---
    operator: equal
    expected: 'foo'
    actual:   'bar'
  ...

1..1
# tests 1
# pass  0
# fail  1
```

Find out more about Tape here: **https://github.com/substack/tape**.

REACT

React and Redux are common partners. The declarative nature of React fits perfectly to Redux's state model, so it's no wonder that people love using these two libraries together. We will be using the official `react-redux` library, which provides bindings between the two libraries.

Find out more about React here: **https://facebook.github.io/react/**.

EXPRESS

Express is a minimalist framework for Node.js. Even though we are building a single page app, it is still a good idea to have a small server actually running in the background to serve the static HTML file and, eventually, our API.

In this chapter, the Express server will perform the following tasks:

1. Serve any file from the `public` directory (this is where our app.js file will live).
2. For any other GET request, just serve the index.html file.
3. Listen on port 3000.

```
const path = require('path');
const express = require('express');
const app = express();

app.use(express.static('public'));

app.get('*', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'public/index.html'));
});

app.listen(3000, () => {
  console.log('`Developing a Redux Edge` notes app listening on port 3000!');
});
```

Find out more about Express here: **http://expressjs.com/**.

Getting started

Now let's move to the next steps.

Implementation

We will now implement each part of our application step-by-step, writing tests along the way. All of this code can be found in the book's Github repository.

Action creators

In our spec we defined which actions we would need for our application to have an acceptable level of interactivity. Let's implement action creators for these actions now.

ADDING

The ability to add a new note is our most essential feature, so let's create the action for it first. As we are following the TDD methodology, we will first write up a test for our action creator:

```
import test from 'tape';
import * as actions from './actions';

test('action creator | addNote :: Create correct action',
  ({ deepEqual, end }) => {

    const actualAction = actions.addNote('Hi', 'id-123', 1);
    const expectedAction = {
      type: 'app/addNote',
      payload: {
        id: 'id-123',
        content: 'Hi',
        timestamp: 1,
      },
    };

    deepEqual(actualAction, expectedAction);
    end();
  }
);
```

We want our action to have the proper `id`, `content`, and `timestamp` payload keys and the correct `app/addNote` `type`. If we run this right now it will obviously fail, as we haven't created our action yet. Luckily the action creator is straight forward:

```
import { v4 } from 'uuid';

export const addNote = (content = '', id = v4(), timestamp = Date.now()) =>
({
  type: 'app/addNote',
  payload: {
    id,
    content,
    timestamp,
  },
});
```

Now our tests pass just fine. One down, four to go!

UPDATING

Since we can create notes now, we should also be able to update them with new content. Here is our test code:

```
test('action creator | updateNote :: Create correct action',
  ({ deepEqual, end }) => {

    const actualAction = actions.updateNote('Hello', 'id-123', 2);
    const expectedAction = {
      type: 'app/updateNote',
      payload: {
        id: 'id-123',
        content: 'Hello',
        timestamp: 2,
      },
    };
    deepEqual(actualAction, expectedAction);
    end();
  }
);
```

You will notice that this code is identical to our `addNote` test code, except for the action `type`. It is fine for tests to contain duplicate logic. We could now write a generic testing method that tests both actions and allows us to pass in a custom `type`, but that would severely hinder our refactoring ability later on if we chose to modify one of the actions. Keeping tests isolated, even if they are more verbose and contain duplicated code, helps us to make sure that each of them can be refactored without much trouble.

Since our test is failing, let's make it pass:

```
export const updateNote = (content, id, timestamp = Date.now()) => ({
  type: 'app/updateNote',
  payload: {
    id,
    content,
    timestamp,
  },
});
```

And that's it! Another action creator done. Now we are able to tell our store that we want to modify an existing note.

REMOVING

If a note is not needed anymore we should be able to remove it from our store. All that we need for that is a proper `type` and the `id` of the note in question:

```
test('action creator | removeNote :: Create correct action',
  ({ deepEqual, end }) => {

    const actualAction = actions.removeNote('id-123');
    const expectedAction = {
      type: 'app/removeNote',
      payload: {
        id: 'id-123',
      },
    };

    deepEqual(actualAction, expectedAction);
    end();
  }
);
```

The action creator to make this test pass is very simple:

```
export const removeNote = (id) => ({
  type: 'app/removeNote',
  payload: { id },
});
```

And with this we have ticked off adding, updating and removing of a note. Now we have two actions left that do not modify a note's data, but that do modify our application's UI state.

OPENING A NOTE

What good is a notes app if you cannot view your notes? There's nothing easier than that, so all we need is another action creator:

```
test('action creator | openNote :: Create correct action',
  ({ deepEqual, end }) => {

    const actualAction = actions.openNote('id-123');
    const expectedAction = {
      type: 'app/openNote',
      payload: {
        id: 'id-123',
      },
    };

    deepEqual(actualAction, expectedAction);
    end();
  }
);
```

Just like earlier in this chapter, you will notice that the code is very similar to the previous test code. You can see two actions with a similar shape, but a different intent. Because of that, our action creator also looks similar, yet the `type` property differs:

```
export const openNote = (id) => ({
  type: 'app/openNote',
  payload: { id },
});
```

This action is going to help show a note when you select one from our list. If we open a note, though, we should also be able to close it.

CLOSING A NOTE

This is the last action we have to implement for now, and it is the simplest of all:

```
test('action creator | closeNote :: Create correct action',
  ({ deepEqual, end }) => {

    const actualAction = actions.closeNote();
    const expectedAction = {
      type: 'app/closeNote',
    };

    deepEqual(actualAction, expectedAction);
    end();
  }
);
```

This does not require any additional information except the correct type. This type is enough to convey the action's intent, so let's make the tests pass once more:

```
export const closeNote = () => ({
  type: 'app/closeNote',
});
```

We now have all of the action creators, so we need to bring some interactivity to our application. If we were to dispatch these actions now, nothing would happen, since we have no reducer set up to handle them. We are going to rectify that now!

Reducers

Without reducers we have no place to store our data and that's a shame for a data-driven application. We are going to model the application's state using three distinct reducers: `byId`, `ids`, and `openNoteId`. It might seem unnecessary at first to create a reducer for a tiny part like `openNoteId`, but it will help keep our reducers small and focused on a single responsibility.

TEST SETUP

All of our reducers will be tested in the same file, which will contain the following imports at the top:

```
import test from 'tape';
import * as reducers from './reducers';
import * as actions from './actions';
import { getMockState } from './testUtils';
```

These will not be repeated throughout the code examples below.

In order to not clutter our tests too much, we are using a tiny `getMockState` helper. It is a simple object with methods on it, which each return a specific state object to be used as a reducer's initial state.

```
// source/store/testUtils.js

export const getMockState = {
  withNoNotes: () => ({
    byId: {},
    ids: [],
    openNoteId: null,
  }),
  withOneNote: () => ({
    byId: {
      'id-123': {
        id: 'id-123',
        content: 'Hello world',
        timestamp: 1,
      },
    },
    ids: [ 'id-123' ],
    openNoteId: 'id-123',
  }),
  // Etc... for all state shapes we need for our tests.
};
```

The reason we are using methods to return a new state objects every time — instead of simply having a large static object — is because we want to protect ourselves from making it possible for tests to affect one another. If a test was to accidentally mutate part of the state inside the helper (and the helper was just a plain mutable JS object), it would result in the following tests working with modified data, probably making those tests fail without a good reason — a debugging nightmare! Using a method to return a new state object each time makes sure our tests are truly isolated from one another.

BYID **REDUCER**

The responsibility of this reducer is to store each created note in an object keyed by its id, so we can easily lookup and modify it. It will have to handle three actions: `addNote`, `upda-teNote`, and `removeNote`. We will test and implement each action individually.

`addNote`

When adding a new note, we want the state to contain a new key, which is the note's id and the note itself as its value:

```
test('reducer | byId :: Handle "addNote" action',
  ({ deepEqual, end }) => {

    const state = getMockState.withNoNotes();

    const actualNextState = reducers.byId(state.byId, actions.addNote('Hello
world', 'id-123', 1));
    const expectedNextState = {
      'id-123': {
        id: 'id-123',
        content: 'Hello world',
        timestamp: 1,
      },
    };

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

The test is straight forward: we compare two states and assert that they are equal.

Handling our `addNote` action now couldn't be easier, and all we have to do is to add the new note to our state:

```
import { merge } from 'ramda';

export const byId = (state = {}, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
      return merge(state, { [payload.id]: payload });
    default:
      return state;
  }
};
```

Note: We are using the `ramda` library here to perform our state mutations. It is a functional programming library that is immutable by default, meaning it will never modify its arguments, but always return new objects. At the end of this chapter we will show some examples how state updates could be done with pure ES6/7 and give a short explanation why a library is useful.

`merge` takes two objects and returns the merged result. The passed in objects will be merged left-to-right, so keys from the second object will overwrite keys from the first object. Our test will pass now as we fulfilled the requirements.

`updateNote`

When a note changes, we want our state to update itself accordingly, replacing the previous version of the note with a new one:

```
test('reducer | byId :: Handle "updateNote" action',
  ({ deepEqual, end }) => {

    const state = getMockState.withOneNote();

    const actualNextState = reducers.byId(state.byId, actions.updateNote('Hi
there', 'id-123', 2));
    const expectedNextState = {
      'id-123': {
        id: 'id-123',
        content: 'Hi there',
        timestamp: 2,
      },
    };

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

You may have noticed that our requirements are pretty much the same as for adding a new note: all we need is for our reducer to replace the value of the property specified by `payload.id` with the new payload. Because of this we can reuse the logic for adding a note and just tell our reducers to handle both actions in the same way:

```
import { merge } from 'ramda';

export const byId = (state = {}, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
    case 'app/updateNote':
      return merge(state, { [payload.id]: payload });
    default:
      return state;
  }
};
```

Just one more action and our `byId` reducer is ready for prime-time!

`removeNote`

When removing a note we want our reducer to delete the property specified by `payload.id` from its state:

```
test('reducer | byId :: Handle "removeNote" action',
  ({ deepEqual, end }) => {

    const state = getMockState.withOneNote();

    const actualNextState = reducers.byId(
      state.byId,
      actions.removeNote('id-123')
    );
    const expectedNextState = {};

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

Sadly we cannot reuse the logic from the previous two actions for this, but the solution is still very easy:

```
import { merge, dissoc } from 'ramda';

export const byId = (state = {}, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
    case 'app/updateNote':
      return merge(state, { [payload.id]: payload });
    case 'app/removeNote':
      return dissoc(payload.id, state);
    default:
      return state;
  }
};
```

We can just use `ramda`'s `dissoc` function for this: it takes a prop and an object and returns a new object without the prop. Great, again this is just what we need! And we have now finished our `byId` reducer. It can now handle all three actions.

`IDS` REDUCER

The `ids` reducer will be used to keep track of all our note ids in an array. We will use this array later when reading the notes from our state. The ids array will specify the order of our notes. As ids are not changing we do not need to handle any update action, since `addNote` and `removeNote` are the only two actions this reducer cares about.

`addNote`

Whenever a new note is added, we need to get its id and add it to the beginning of our ids array.

```
test('reducer | ids :: Handle "addNote" action',
  ({ deepEqual, end }) => {

    const state = getMockState.withNoNotes();
```

```
      const actualNextState = reducers.ids(
        state.ids,
        actions.addNote("Hi", "id-123")
      );
      const expectedNextState = [ 'id-123' ];

      deepEqual(actualNextState, expectedNextState);
      end();
    }
  );
```

The above test is very straight forward and so is our implementation that makes it pass:

```
import { prepend } from 'ramda';

export const ids = (state = [], { type, payload }) => {
  switch (type) {
    case 'app/addNote':
      return prepend(payload.id, state);
    default:
      return state;
  }
};
```

`prepend`, as you might have already guessed, prepends `payload.id` to a copy of `state`. Great! Just one more action for this reducer:

`removeNote`

When a note gets removed, you need to make sure to remove its id from the ids array as well, so that you don't accidentally try to read data for a non-existing id later on.

```
test('reducer | ids :: Handle "removeNote" action',
  ({ deepEqual, end }) => {

    const state = getMockState.withOneNote();

    const actualNextState = reducers.ids(
      state.ids,
      actions.removeNote("id-123")
    );
    const expectedNextState = [];

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

Our test case again is very easy to understand: When we have one id in our initial state and we remove a note with the same id, we want our new state to be an empty array. Our updated reducer that passes this test looks like this:

```
export const ids = (state = [], { type, payload }) => {
  switch (type) {
    case 'app/addNote':
      return prepend(payload.id, state);
    case 'app/removeNote':
      return without(payload.id, state);
    default:
      return state;
  }
};
```

`without` takes in a value and an array and will return a new array that does not contain the value. This fits our use-case perfectly and with that we have finished our `ids` reducer.

`OPENNOTEID` REDUCER

Our `openNoteId` reducer takes care of storing which note is currently being displayed. Because of this it has to only store a single value - an id. There are four actions that can cause a change to the reducer's state:

- `openNote`
- `closeNote`
- `addNote`
- `removeNote`

Let's add them one by one:

`openNote`

Whenever a user wants to look at a note, our application will have to dispatch an `openNote` action. When this action is dispatched, we expect that our `openNoteId` state will then contain that id:

```
test('reducer | openNoteId :: Handle "openNote" action',
  ({ equal, end }) => {

    const state = getMockState.withNoOpenNotes();

    const actualNextState = reducers.openNoteId(
      state.openNoteId,
      actions.openNote("id-123")
    );
    const expectedNextState = 'id-123';
    equal(actualNextState, expectedNextState);
    end();
  }
);
```

If the above sounded simple to you, you're right! We have to do nothing else than to return the `payload`'s `id` property:

```
export const openNoteId = (state = null, { type, payload }) => {
  switch (type) {
    case 'app/openNote':
      return payload.id;
    default:
      return state;
  }
};
```

When splitting up reducers in a way that they only serve a single purpose, their internals can often become trivial, as can be seen with this example.

addNote

The way our application should work is that if a user adds a new note, it will instantly display that note, even if another note is open. Our test is similar to the one for `openNote`: we expect the id of the `addNote` action to become the new state:

```
test('reducer | openNoteId :: Handle "addNote" action',
  ({ equal, end }) => {

    const state = getMockState.withNoOpenNotes();

    const actualNextState = reducers.openNoteId(
      state.openNoteId,
      actions.addNote("Hi", "id-123")
    );
    const expectedNextState = 'id-123';

    equal(actualNextState, expectedNextState);
    end();
  }
);
```

This again looks very familiar, doesn't it? We can just reuse the simple logic we added for `openNote`:

```
export const openNoteId = (state = null, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
```

```
    case 'app/openNote':
      return payload.id;
    default:
      return state;
  }
};
```

Both actions will now trigger an update to the currently open note!

`closeNote`

When a user is done editing a note, they should be able to close it. We already added the `closeNote` action for this purpose, so let's add it to our reducer now. We will write our test first again:

```
test('reducer | openNoteId :: Handle "closeNote" action',
  ({ equal, end }) => {

    const state = getMockState.withOneNote();

    const actualNextState = reducers.openNoteId(
      state.openNoteId,
      actions.closeNote("id-123")
    );
    const expectedNextState = null;

    equal(actualNextState, expectedNextState);
    end();
  }
);
```

Whenever we receive a `closeNote` action we just want our new state for `openNoteId` to become `null`. There's nothing easier than that:

```
export const openNoteId = (state = null, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
    case 'app/openNote':
      return payload.id;
    case 'app/closeNote':
      return null;
    default:
      return state;
  }
};
```

We don't have to worry about the payload of the action for this one because we just want to reset our state. It in fact doesn't have payload data, but even if it did - we wouldn't care about it in this reducer.

`removeNote`

The last action we have to take care of is `removeNote`. When a user removes a note that is currently open, then the system has to take care of closing it automatically. We could be falling back to the next note in line, but for now we will only close the removed note and not do anything else. So we expect our state, just like for `closeNote`, to become `null` when the `removeNote` action is being dispatched:

```
test('reducer | openNoteId :: Handle "removeNote" action',
  ({ equal, end }) => {

    const state = getMockState.withOneNote();

    const actualNextState = reducers.openNoteId(
      state.openNoteId,
      actions.removeNote("id-123")
    );
    const expectedNextState = null;

    equal(actualNextState, expectedNextState);
    end();
  }
);
```

And just like with `closeNote`, we do not care about any of the action's payload, we just want to reset our state. Because of this we just handle the action together with `closeNote`:

```
export const openNoteId = (state = null, { type, payload }) => {
  switch (type) {
    case 'app/addNote':
    case 'app/openNote':
      return payload.id;
    case 'app/removeNote':
    case 'app/closeNote':
      return null;
    default:
      return state;
  }
};
```

That's it! We have all of our reducers properly set up and tested. We have state in our reducers, but how do we get it out? In the next chapters we will focus on exactly that. But before we do so, here is some additional information about how to modify reducer state without a handy library.

Modifying state without a library

If we don't want to use a library such as ramda or lodash, we could write the above code as pure ES6/7. Here are some examples:

```
// `byId` reducer
// ramda
return merge(state, { [payload.id]: payload });
// pure ES6
return Object.assign({}, state, { [payload.id]: payload };
// pure ES7
return { ...state, [payload.id]: payload };

// ramda
return dissoc(payload.id, state);
// pure ES6
const nextState = Object.assign({}, state);
delete nextState[payload.id];
return nextState
//pure ES7
const nextState = { ...state };
delete nextState[payload.id];
return nextState

// `ids` reducer
// ramda
return prepend(payload.id, state);
// pure ES6
return [payload.id].concat(state);
// pure ES7
return [payload.id, ...state];

// ramda
return without(payload.id, state);
// pure ES6/7
return state.filter(id => id !== payload.id);
```

While the first example is even a little bit shorter in ES7, the second one instantly shows how a utility library can help make our code cleaner and more readable. Furthermore - as mentioned before - `ramda` is immutable by default, so we do not have to think about whether we accidentally are mutating an object - we get a fresh copy each time. In addition to that, using functions like `merge`, `dissoc`, `prepend`, `without` etc. make the intent of our code very clear and easy to understand. Ultimately it is up to you what you prefer. For the sake of readability and ease we chose to stick with `ramda` for the book.

Selectors

Ok, now that we have the state transformation logic done and dusted, it is time to create a way we can deliver slices of our state from Redux to React. In order to do that, we need to write selectors.

Looking back at our brief, we know that we have two component-views to worry about: Notes List and Note Detail.

NOTES LIST SELECTOR (GETNOTES)

Starting with Notes List, this component will need to present a list of note thumbnails along with the beginning of each note's content.

This means that what we need to get from the state is an array populated with individual notes. That way, once we get the state into our components, we can write something like: `notes.map((note) => (<div> /* Note markup */ </div>))` to generate the Notes List view.

Ok, so we need a selector that will get all of the notes. Let's call it `getNotes`, and describe it in test form:

```
// source/store/selectors.test.js

import test from 'tape';
import * as selectors from './selectors';
import { getMockState } from './testUtils';

test('selector | getNotes :: Return empty array if state contains no notes',
  ({ deepEqual, end }) => {

    const state = getMockState.withNoNotes();

    const actualSelection = selectors.getNotes(state);
    const expectedSelection = [];

    deepEqual(actualSelection, expectedSelection);
    end();
  }
);

test('selector | getNotes :: Return array of note objects if state contains
any notes',
  ({ deepEqual, end }) => {

    const state = getMockState.withOneNote();

    const expectedSelection = [
      {
        id: 'id-123',
        content: 'Hello world',
        timestamp: 1,
      },
    ];
    const actualSelection = selectors.getNotes(state);

    deepEqual(actualSelection, expectedSelection);
    end();
  }
);
```

Now that we have our tests, how do we actually write the selector? First, let's look at our current state shape:

```
// State shape representing two notes present, with note `id-456`
// currently being open:
{
    byId: {
      'id-123': {
        id: 'id-123',
        content: 'Hello world',
        timestamp: 1,
      },
      'id-456': {
        id: 'id-456',
        content: 'Hi globe',
        timestamp: 2,
      },
    },
    ids: [ 'id-123', 'id-456' ],
    openNoteId: 'id-456',
  }
```

Looking at the above, it becomes clear that we can't simply grab a part of the state tree directly — nowhere in our state do we actually *have* an array of notes. This means we have to *derive* the state.

Derived state is state that is extrapolated from the main store state. You will see this pattern used quite a bit.

In this case, what we have to do is map over the `ids` array and use it to generate a new, populated array from the `byId` state slice.

```
// source/store/selectors.js

export const getNotes = (state) =>
  state.ids.map((id) => state.byId[id]);
```

Great. This makes both our tests pass, because `Array.map` will simply return an empty array if the `ids` state slice doesn't contain anything.

The Notes List view only requires the `getNotes` selector, which means we are free to start working on Note Detail.

NOTE DETAIL SELECTORS (GETNOTE, GETOPENNOTEID)

If you refer back to the mockups, you can see that you will need the full note object, including content, timestamp, and id. We know that this will have to be the note whose ID is currently stored inside the `openNoteId` state slice.

Now, this could be either written as a single more specific selector (`getOpenNote`) or two tiny selectors (`getOpenNoteId`, `getNote`) that will work in tandem to return the desired note.

For the sake of modularity and reuse, let's go with the second option. That way, you are able to prepare generic selectors ahead of time, and compose them together just before you pass them to your components (remember, they are just functions!).

You'll see how exactly that is done in the last section of this chapter. Now, let's start implementing our two generic selectors.

```
// source/store/selectors.test.js

// ...

test('selector | getOpenNoteId :: Return null if state doesn\'t have open
note set',
  ({ equal, end }) => {

    const state = getMockState.withNoNotes();

    const actualSelection = selectors.getOpenNoteId(state);
    const expectedSelection = null;

    equal(actualSelection, expectedSelection);
    end();
  }
);

test('selector | getOpenNoteId :: Return note id if state has open note set',
  ({ equal, end }) => {

    const state = getMockState.withOneNote();

    const actualSelection = selectors.getOpenNoteId(state);
    const expectedSelection = 'id-123';

    equal(actualSelection, expectedSelection);
    end();
  }
);

test('selector | getNote :: Return null if state doesn\'t contain a note
with supplied id',
  ({ equal, end }) => {

    const state = getMockState.withTwoNotes();

    const actualSelection = selectors.getNote(state, 'id-999');
    const expectedSelection = null;

    equal(actualSelection, expectedSelection);
    end();
  }
);

test('selector | getNote :: Return note object if state contains note with
passed id',
  ({ deepEqual, end }) => {

    const state = getMockState.withTwoNotes();

    const actualSelection = selectors.getNote(state, 'id-123');
    const expectedSelection = {
      id: 'id-123',
      content: 'Hello world',
      timestamp: 1,
    };

    deepEqual(actualSelection, expectedSelection);
    end();
  }
);
```

We know the initial and empty states of the `openNoteId` state slice are `null` (as we want), and when a note is opened it simply contains the string. This means that `getOpenNoteId` just returns a piece of the state, which is simple enough.

```
// source/store/selectors.js

// ...

export const getOpenNoteId = (state) =>
  state.openNoteId;
```

Our last selector, `getNote` is a little different, because it doesn't just take the state, but also an `id` as a second argument. This makes it nicely generic and allows us to eventually reuse it with arbitrary notes, not just the currently open note.

```
// source/store/selectors.js

// ...

export const getNote = (state, id) =>
  state.byId[id] || null;
```

The selectors are now completed. Let's finish off the MVP by using them inside our components!

Connecting to UI

And now, for the last stretch — finally connecting React and Redux together!

We will start by building a static version of our view-components: `NotesList` and `NoteDetail`. These two will be joined together in a root component called App, which will in turn be passed to `ReactDOM.render`.

Once the static stuff is done, we will hook it all up to back to Redux and finally get the party started!

Note that as we are building the React components, we will omit propType validation and component testing. This is done to save you from reading a whole bunch of verbose code that doesn't really add any additional insight. After all, this is a book about Redux, not React or any particular UI library. Additionally, we also won't show the details of the style object. Just assume that in it contains the correct inline styles for each element.

If you're interested in looking at the code without any omissions, the GitHub repo (**https://github.com/arturmuller/developing-a-redux-edge**) contains the app in its entirety.

STATIC COMPONENTS

Starting with `NotesList`, let's see what we have:

```
// source/components/NotesList/index.jsx

import React, { PropTypes } from 'react';
import * as style from './style';

const NotesList = ({ notes, openNoteId, addNote, openNote }) => (
  <div style={style.wrapper}>
    <button
      style={style.addNoteButton}
```

```
          onClick={addNote}
          >
          Add Note
        </button>
        {(notes.length === 0)
          ? <div style={style.blankslate}>No notes</div>
          : notes.map((note) => (
              <button
                key={note.id}
                style={(note.id === openNoteId)
                  ? { ...style.note, ...style.selected }
                  : style.note
                }
                onClick={() => openNote(note.id)}
                >
                {note.content === ''
                  ? <span style={style.newNoteLabel}>New note...</span>
                  : note.content
                }
              </button>
          ))
        }
      </div>
    );

    export default NotesList;
```

This is pretty standard React stuff. Notice how the component is written in a pure functional style, instead of the more common classical approach (`class NotesList extends React.Component`) or by using the old-school `React.createClass`.

Functional components are beautifully compact and terse, but cannot have internal state or methods. This is juts fine by us; all of our state lives inside of Redux (as opposed to local component state), and our actions are defined separately as actionCreators (as opposed to methods on the component class)!

Now, let's look at `NoteDetail`:

```
    // source/components/NoteDetail/index.jsx

    import React, { PropTypes } from 'react';
    import * as style from './style';

    const NoteDetail = ({ note, removeNote, closeNote, updateNote }) => (
      <div style={style.wrapper}>
        {!note
          ? <div style={style.blankslate}>No note is open</div>
          : <div style={style.note}>
              <div style={style.date}>
                {new Date(note.timestamp).toLocaleString()}
              </div>
              <textarea
                autoFocus
                key={note.id}
                style={style.textarea}
                onChange={(event) => updateNote(event.target.value, note.id)}
                placeholder="New note..."
                value={note.content}
                />
              <div style={style.row}>
                <button
                  onClick={() => removeNote(note.id)}
                  style={{ ...style.button, ...style.danger }}
                  >
                  Remove
                </button>
                <button
                  onClick={closeNote}
                  style={style.button}
                  >
                  Close
                </button>
              </div>
            </div>
        }
      </div>
    );

    export default NoteDetail;
```

And finally App:

```
// source/components/App/index.jsx

import React from 'react';
import NotesList from '../NotesList';
import NoteDetail from '../NoteDetail';
import * as style from './style';

const App = () => (
  <div style={style.wrapper}>
    <NotesList />
    <NoteDetail />
  </div>
);

export default App;
```

There is nothing much different about the two components above. `NoteDetail` renders markup relevant to its use-case just as `NoteDetail` did, and `App` wraps the two together so that we have a 'root' component we can pass to ReactDOM.

Let's do that now:

```
// source/index.jsx

import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';

ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

Notice how this is the main index file. Finally, we have reached the end of the tunnel!

Unfortunately the app doesn't really do much at the moment. If you try to run it now, you'd get a whole bunch of errors because no data is actually being passed from the store to the components. Let's fix that!

CONNECTED COMPONENTS

As we have learned in Chapter 2, to get everything joined up, we first need to make the store available to the component hierarchy through the `Provider` from `react-redux`.

Only one `Provider` is necessary, and ideally it should be as high-up in your component tree as possible. This is done to avoid trying to `connect` a component outside of `Provider`'s scope.

We will add the `Provider` directly in the main `index.jsx` file. This way, as long as we consider `App` the root of the app, we will always be able to `connect` any component.

```
// source/index.jsx

// ...
import { Provider } from 'react-redux';
import configureStore from './store';

ReactDOM.render(
  <Provider store={configureStore()}>
    <App />
  </Provider>,
  document.getElementById('app')
);
```

Now, let's return to our view-components.

If you look back at our component's dependencies (as defined through destructuring the props parameter), we will see exactly what data we need to retrieve for the component.

Some of that data is state, so it will have to be retrieved through a selector. The other data are actions. These need to be wrapped in `store.dispatch` before being passed to the components to actually do anything. (Remember, action creators are just functions that return a descriptor object, since it is `store.dispatch` that actually makes stuff happen!)

The sate and actions above can be achieved with the `connect` higher order component. To quickly recap, `connect` accepts the `mapStateToProps` function as the first argument and `mapDispatchToProps` as the second argument.

`mapStateToProps` will be passed the entire Redux state, and its job is to select only the parts that are interesting for the component and returning them as an object that will be merged with props.

`mapDispatchToProps` can also be a function that receives `dispatch` as an argument, and its job is to return on an object of dispatch-ready actions. Again, this object will be merged into props.

Most of the time all you need to do with `mapDispatchToProps` is wrap dispatch around actions like so:

```
const mapDispatchToProps = (dispatch) => ({
  action: (...args) => dispatch(actionCreator(...args)),
})
```

Because this is such a common pattern, Redux offers a shorthand: just pass an object of action creators instead of a function and the above will be done for you.

(If you're not sure what is happening above, Chapter 2 offers more detail about the `connect` function.)

```
// source/components/NotesList/index.jsx

// ...
import * as actionCreators from '../../store/actions';
import * as selectors from '../../store/selectors';

const NotesList = ({ notes, openNoteId, addNote, openNote }) => (
  // ...
);

const mapStateToProps = (state) => ({
  notes: selectors.getNotes(state),
  openNoteId: selectors.getOpenNoteId(state),
});

export default connect(mapStateToProps, actionCreators)(NotesList);
```

We have imported all selectors and action creators we have prepared in the previous chapters using the `*` notation. This essentially bundles all named exports of a given module into an object with the name you have assigned by `as <name>`.

For action creators this means we are ready — we simply pass the entire imported object to `connect` as the second argument and let the `mapDispatchToProps` shorthand take over wrapping everything in `dispatch`.

For the first argument, we will create a component-specific `mapStateToProps` function that will combine our generic selectors, and essentially just combine their input into a single object.

```
// source/components/NoteDetail/index.jsx

// ...

import * as actionCreators from '../../store/actions';
import * as selectors from '../../store/selectors';

// ...

const mapStateToProps = (state) => ({
  note: selectors.getNote(state, selectors.getOpenNoteId(state)),
});

export default connect(mapStateToProps, actionCreators)(NoteDetail);
```

Notice how we're composing our generic selectors slightly differently to `NotesList` here. Instead of assigning the results each to its individual key, we take the result of `getOpenNoteId()` and pass it as an argument to `getNote()`.

Conclusion

Although these are still fairly simple use-cases, they do illustrate that because selectors are just functions, working with them is very versatile. You can compose and rearrange them as you see fit. Although these are still fairly simple use-cases, they do illustrate that because selectors are just functions, working with them is very versatile. You can compose and rearrange them as you see fit.

We now have a fully functional MVP version of our app! In the next chapter we will add server-side persistence to our application using asynchronous actions and the `redux-thunk` middleware.

5 Syncing Data with the Server

We have an application, which can be used to create and edit basic notes. But when we refresh the browser, all our data is lost and we start fresh. In this chapter we will focus on adding server-side persistence to our application using asynchronous actions and the `redux-thunk` middleware, since it is the most basic building block for asynchronous actions in Redux. Why `redux-thunk`? Simply because it is the most generic way to do asynchronous actions in Redux. Originally it was part of the Redux core, but it has since been extracted into its own package. Later in this chapter we will look at alternative approaches.

Note: You can find the complete source code for the sample application at **https://github.com/arturmuller/developing-a-redux-edge**.

Application specification

Let's walk through all of the specs of our application.

Actions

Our original actions have all been synchronous and this has to change. In addition to that we will also need a new action to retrieve all of our notes on the initial application render.

ADDING A NOTE

This action will add a new note to our server's state.

UPDATING A NOTE

This action will update a note's state on the server.

REMOVING A NOTE

This action will remove a note from our server's state.

FETCHING A LIST OF NOTES

This is a new action that will retrieve the current list of notes from our server so we can populate our state with it.

Server

We provide a basic server implementation for this example, but we will be treating it as a black box, or a third party service. You can of course inspect the server code in the repository if you wish, but creating the server is not part of this chapter.

The following calls are recognized by the server:

- **GET /api/notes** should be used for reading all notes currently on the server. It responds with an array of note objects.
- **POST /api/notes** should be used to create a new note. It responds with the new note.
- **DELETE /api/notes/:id** should be used for removing notes. It responds with the deleted note.
- **PUT /api/notes/:id** should be used to update a note. It responds with the updated note.

All of our notes are going to be stored on our server, but only in memory. For the sake of keeping things simple we will not introduce any kind of database and just use a basic object to store the notes. This means that our notes will not persist across server restarts.

Implementation

Let's move on by interacting with the API.

Api module

In order to interact with our API we will need some sort of interface. We could just keep all of the requests directly without our action creators, but that would get messy really fast. Because of that we are opting for creating a separate API module that will contain all of our API calls. We will be using `isomorphic-fetch` for making requests. This library enables you to use the upcoming standard `fetch` API. In addition to that we know that we will have to load a list of notes. We also know that we have a `byId` and `ids` reducer. Instead of extracting all note ids and keying the objects on our own, we are going to use a library called `normalizr` to take care of that for us. It is primarily used for processing nested responses, but it also works well for a basic use case like this. At the time of writing, the library weighs `18.3kb` minified, which is quite hefty if we were only going to use it for something really basic like this. But over the course of this book we will extend our application more and will have a greater need for `normalizr`. Because of that, we are going to introduce it right from the start. As always: it is a judgement call whether or not you want to use an external library.

FETCHING NOTES

The first method we will focus on is going to fetch a list of notes. Let's start by writing some tests for it:

```
// source/utils/api.test.js
// imports omitted for brevity

test('api | notes.fetch ::', (t1) => {
  const NOTE_ONE = {
    id: 123,
    content: 'testing',
    timestamp: 1,
  };
  const NOTE_TWO = {
    id: 456,
    content: 'testing too',
    timestamp: 1,
  };
  const DUMMY_CREATE_RESPONSE = [ NOTE_ONE, NOTE_TWO ];

  t1.test('Returns notes and noteIds', ({ deepEqual, end }) => {
    fetchMock.mock('/notes', DUMMY_CREATE_RESPONSE);
    api.notes
      .fetch()
      .then(({ notes, noteIds }) => {
        const expedtedNotes = {
          [NOTE_ONE.id]: NOTE_ONE,
          [NOTE_TWO.id]: NOTE_TWO,
        };
        const expectedNoteIds = [ 123, 456 ];

        deepEqual(noteIds, expectedNoteIds);
        deepEqual(notes, expedtedNotes);

        end();
      });
  });

  t1.test('Gets rejected with an error containing the status text if the sta-
tus is not 2xx', ({ equal, end }) => {
    testErrorRejection({
      path: '/notes',
      equal,
      func: () => api.notes.fetch(),
    })
```

```
      .then(() => end());
  });

  t1.test('Teardown', ({ end }) => {
    fetchMock.restore();
    end();
  });
});
```

We are using `fetch-mock` in our tests, which is a library that allows us to mock requests to certain endpoints and return arbitrary data as a result. This is perfect for testing our API code without hitting any API! We are relying on the fact that our API behaves as we expect it to and we can treat it as a black box, without testing the actual responses.

Since `fetch` doesn't reject its promise by default if it receives status code that is not between 200 and 300, we are testing for that too, because we will have to implement it on our own. Since we should test this for all methods, we are using the `testErrorRejection` helper. The code for this helper can be found in `source/utils/testUtils.js`.

The "Teardown" test is a way to simulate the "after" method, which you might know from testing libraries like `mocha`. We need to make sure that we clean up after ourselves to not affect other tests.

Since this test will obviously fail for now, we will go right ahead and implement our first API method. Here is the code:

```
import 'isomorphic-fetch';
import { normalize, Schema, arrayOf } from 'normalizr';

const notes = new Schema('notes');

export const toJson = (res) => res.json();

export const checkStatus = (res) => {
  const { status } = res;
  if (status >= 200 && status < 300) {
    return res;
  }

  return Promise.reject(new Error(res.statusText || res.status));
};

export const normalizeNotesList = (data) => normalize(data, arrayOf(notes));

export const returnNotesAndIds = ({ entities: { notes }, result: noteIds })
=> ({
  notes,
  noteIds,
});
```

```
export const fetchJson = (url, options = {}) => (
  fetch(url, {
    ...options,
    headers: {
      ...options.headers,
      'Accept': 'application/json',
      'Content-Type': 'application/json',
    },
  })
  .then(checkStatus)
  .then(toJson)
);

export default {
  notes: {
    fetch() {
      return fetchJson('/notes')
        .then(normalizeNotesList)
        .then(returnNotesAndIds);
    },
  },
};
```

We import `isomorphic-fetch` without assigning it to a variable, since it will patch the global `fetch` object.

First, we have to create our `notes` schema, which - as you can see - is really straight forward. As our notes currently have no nested children, this is all we have to do at the moment to use `normalizr`.

The `toJson` function is a basic utility we will reuse for all our `fetch` calls: it converts the response stream to JSON. This is part of the `fetch` specification, since `fetch` always returns a stream, which can be converted into different output formats. More info about this can be found on the **Mozilla developer network**.

As noted before, a `fetch` promise is not going to be rejected for a bad status code. It will only be rejected if the fetching failed, e.g. if the connection could not be established. Because of this we have the `checkStatus` function that is going probe the response status and maybe reject the promise.

Because all of our requests are going to interact with a JSON API, we add a wrapper function around `fetch`, called `fetchJson`, which is going to call `checkStatus` and `toJson` for us, and will also set the required headers automatically. You can still override them if you have to. This will keep the code a bit more DRY.

In order to fetch the notes, we issue a request to the `/notes` endpoint of our API. The `fetch` API takes two arguments, a URL or `Request` object, and an *optional* options object. Again, more information on this can be found on the **fetch manual page of the MDN**.

After our response passes `checkStatus` and `toJson`, we still have to process the data using `normalizr`. The `normalize` methods accepts `data` and `schema` parameters. In our case we know that the API response is going to be an array of notes, so we wrap our schema in `arrayOf`. `normalizr` needs this information in order to properly process our response. It will create an object with an `entities` and a `result` key. `entities` will be an object with a `notes` property, taken from the name of our schema. The value of that property will be an object with all of the notes keyed by id - exactly what we need for our `byId` reducer! The `result` will contain an ordered list of note ids, which we can re-use for our `ids` reducer. In order to make our application less dependent on the shape of the `normalizr` result, for now we are going to transform it again into an object only containing `notes` and `noteIds` properties. This will correspond to the values of `entities.notes` and `result` respectively.

And that's it! We now have the means to fetch all of the notes stored on our server. Before we actually add this functionality to our application, let's implement the rest of our API module.

ADDING A NOTE

Take some content, post it to our API, and receive a new note. That's all this method has to do, and here is the test:

```
// source/utils/api.test.js
// imports and previous tests omitted for brevity
test('api | notes.add ::', (t1) => {
  const NOTE = {
    id: 123,
    timestamp: 1,
  };

  t1.test('Returns a new note from the api', ({ deepEqual, end }) => {
    fetchMock.mock('/notes', 'POST', (url, options) => {
      const { content } = JSON.parse(options.body);
      return {
        ...NOTE,
        content,
      };
    });

    api.notes
      .add('testContent')
      .then((note) => {
        const expectedNote = {
          ...NOTE,
          content: 'testContent',
        };
        deepEqual(note, expectedNote);
        end();
      });
  });

  t1.test('Gets rejected with an error containing the status text if the sta-
tus is not 2xx', ({ equal, end }) => {
    testErrorRejection({
      path: '/notes',
      equal,
      method: 'POST',
      func: () => api.notes.add('test'),
    })
      .then(() => end());
  });

  // Teardown test removed for brevity
});
```

We can reuse the `testErrorRejection` utility now and make sure that the method behaves properly. It should call our API with a JSON object that contains a `content` property and passes through the resulting new note:

```
// source/utils/api.js
// imports, functions and parts of the export omitted for brevity
export default {
  notes: {
    add(content) {
      return fetchJson(
        '/notes',
        {
          method: 'POST',
          body: JSON.stringify({ content }),
        }
      );
    },
  },
};
```

We again use our `fetchJson` helper. This time we have to tell `fetch` that we want to make a POST request and what the `body` is going to be. This will already make our tests pass. Two down, two to go - we are making progress!

UPDATING A NOTE

Being able to create notes on the server, but not having the ability to update them would be kind of useless, so updating a note on the server is up next. Here is the test:

```
// source/utils/api.test.js
// imports and previous tests omitted for brevity
test('api | notes.update ::', (t1) => {
  const NOTE = {
    id: 123,
    timestamp: 1,
  };

  t1.test('Returns an updated note from the api', ({ deepEqual, end }) => {
    fetchMock.mock('/notes/123', 'PUT', (url, options) => {
      const { content } = JSON.parse(options.body);
      return {
        ...NOTE,
        content,
      };
    });

    api.notes
      .update(123, 'testContent')
      .then((note) => {
        const expectedNote = {
          ...NOTE,
          content: 'testContent',
        };
        deepEqual(note, expectedNote);
        end();
      });
  });

  t1.test('Gets rejected with an error containing the status text if the sta-
tus is not 2xx', ({ equal, end }) => {
    testErrorRejection({
      path: '/notes/123',
      equal,
      method: 'PUT',
      func: () => api.notes.update(123, 'test'),
    })
      .then(() => end());
  });

  // Teardown test removed for brevity
});
```

As you can see, this test very closely resembles the test for `api.notes.add`. The only difference is the URL and the method. We can use the same expectations, because we only want the content to be modified. Given the above, you might already expect our actual code to also look very similar to the add method and you're right about that:

```
// source/utils/api.js
// imports, functions and parts of the export omitted for brevity
export default {
  notes: {
    update(id, content) {
      return fetchJson(
        `/notes/${id}`,
        {
          method: 'PUT',
          body: JSON.stringify({ content }),
        }
      );
    },
  },
};
```

A different URL, a different method, and updated method parameters - that's it! the heavy lifting is again done by `fetchJson`.

REMOVING A NOTE

The last API method on our list is removing a note. It will take an `id` and issue a request to our removal endpoint. The test looks similar to our other API tests:

```
// source/utils/api.test.js
// imports and previous tests omitted for brevity
test('api | notes.delete ::', (t1) => {

  t1.test('Removes a note from the api', ({ equal, end }) => {
    fetchMock.mock('/notes/123', 'DELETE', 200);

    api.notes
      .delete(123)
      .then((res) => {
        equal(res, '');
        end();
      });
  });

  t1.test('Gets rejected with an error containing the status text if the sta-
tus is not 2xx', ({ equal, end }) => {
    testErrorRejection({
      path: '/notes/123',
      method: 'DELETE',
      equal,
      func: () => api.notes.delete(123),
    })
      .then(() => end());
  });

  // Teardown test removed for brevity
});
```

This time there is not much to compare, since our delete endpoint will only return an empty string and a `200 OK` response. But that is something you can test, since the implementation will throw for anything except a 2xx status. This is very straight forward:

```
  // source/utils/api.js
  // imports, functions and parts of the export omitted for brevity
  export default {
    notes: {
      delete(id) {
        return fetch(`/notes/${id}`, { method: 'DELETE' })
          .then(checkStatus)
          .then((res) => res.text());
      },
    },
  };
```

All we have to do to satisfy our test is to send a `DELETE` request to `/notes/ID`. One difference to the previous API requests is that we cannot parse our response as JSON, because our removal endpoint only returns an empty string. Because of this we are parsing the response as text.

We have now finished the whole API module. It is now ready to be used within our action creators.

Action creators

Now that our API module is prepared and ready for prime time, we have to update our action creators to support the asynchronous nature of the HTTP requests. For this, `redux-thunk` will be our weapon of choice. In order to test our actions easily, we will introduce a mock store into our tests, using `redux-mock-store`. It allows you to create a store, apply middleware to it and expect what kind of actions get dispatched, without having to simulate any of the store's or middleware's behavior on your own.

NEW `META` AND `ERROR` ACTION SHAPE PROPERTIES

For our actions we will be using `meta` and `error` properties. The `meta` field is designated to contain meta information about the action and was popularized by the **FSA spec**. It's the perfect place for information regarding the state of our request. We could also choose different action types like `app/fetchNotesSuccess` or `app/fetchNotesFailure` to represent different states of our asynchronous flow, but keeping the same action type and updating the `meta` property is more fitting for us and allows for better action grouping. The `error` field signals whether or not an error occurred. Using this spec or convention is obviously a subjective decision. For now we will stick with it though.

TEST HELPERS

Our asynchronous action creators are supposed to return a function to be used by `redux-thunk`, dispatch an action when the action begins, and then one after it succeeds or fails. Knowing that, we can create some helper functions: `assertFirstDispatchedAction`, `assertLastDispatchedAction` and `assertReturnsFunction` are going to help us make assertions around dispatching and the initial return value of the action creator. `createErrorAction` and `createSuccessAction` will reduce the amount of code we have to write to create the correct actions for our assertions. The source for these functions can be found at the top of the file located at `source/store/actions.test.js`. Using these will help reduce the boiler plate for our tests.

Since our tests are going to be rather similar, we will be using another helper called `testAsyncAction`, which will provide the test skeleton for all of the tests. Given that this one is important, let's walk though it right now:

```
// source/store/actions.test.js
const testAsyncAction = ({ test }, options) => {
  const {
    path,
    method,
    successMockReturn,
    successPayload,
    failureMockReturn,
    func,
    type,
  } = options;

  test('Setup', ({ end }) => {
    fetchMock.mock(path, method, successMockReturn);
    end();
  });

  test('Returns a function', (test) => {
    assertReturnsFunction(test, func);
  });

  test('Dispatches initial action', (test) => {
    assertFirstDispatchedAction({ ...test, func, type });
  });

  test('Dispatches correct action on success', (test) => {
    const expectedAction = createSuccessAction(type, successPayload);
```

```
      assertLastDispatchedAction({ ...test, func, expectedAction });
    });

    test('Dispatches correct action on failure', (test) => {
      fetchMock.reMock(path, method, failureMockReturn);
      const expectedAction = createErrorAction(type);
      assertLastDispatchedAction({ ...test, func, expectedAction });
    });

    test('Teardown', ({ end }) => {
      fetchMock.restore();
      end();
    });
  };
```

It is a basic function that will receive a `tape` object as its first argument and some config as the second. Each test will have to set up a mock value for the `fetch` API, for the given `path` and `method`. It also has to provide the desired return value using `successMockReturn`. This value can be one of the many possible options supported by `fetch-mock`. We then have to assert that a function is returned, because this is the base for us facilitating asynchronous actions using `redux-thunk`. Next up are tests to assert that the correct functions are dispatched. They themselves call our previously introduced helpers for asserting dispatches.

In the end, we have to restore the mock values for `fetch` to clean up after ourselves.

ASYNCHRONOUS ACTION HELPERS

It also makes sense for us to introduce a few helper functions for our actual actions. All actions will follow a specific set of steps and shapes. They will start out with an initial action, which will be followed by either a success or a failure action. Given this, we will use four helper functions: `startAction`, `successAction`, `failureAction`, and `asyncAction`. It might seem like a bit of a set up to go through, but they will massively reduce the amount of code we have to write later on. Let’s look at these before we start implementing out action creators.

startAction

The `startAction` helper will create an action creator to dispatch the initial action. So when our requests starts, `meta.done` will be `false`. This is all we need for an initial action. For our purpose, it will not require any kind of payload.

```
  // source/store/actionUtils.js
  export const startAction = (type) => () => ({
    type,
    meta: {
      done: false,
  },
});
```

successAction

When our requests succeed, we will want to dispatch a success action. This action will need to have `meta.done` set to `true` in order to signal that we are done with our request. It will also not contain an `error` property. This way we can correctly determine that the request was a success. Since we will want to pass along some data, the returned function takes a `payload` object that will be attached to the returned action.

```
// source/store/actionUtils.js
export const successAction = (type) => (payload) => ({
  type,
  payload,
  meta: {
    done: true,
  },
});
```

FAILUREACTION

In case our requests fails for whatever reason, we need to inform our application about that too. `meta.done` is going to be set to `true` for this case too, but `error` will be set to `true` as well. In addition to that, any potentially passed along error will be set as the `payload`.

```
// source/store/actionUtils.js
export const failureAction = (type) => (error) => ({
  type,
  payload: error,
  error: true,
  meta: {
    done: true,
  },
});
```

ASYNCACTION

In addition to the previous helper functions we need a way to stitch them together into a meaningful sequence. We could of course have inlined all of the above functionality into a single helper function and just use that, but separating things into the three synchronous and one asynchronous helper not only gives us the benefit of cleaner functions, but also allows us to test our reducers easier later on. We are able to remove our async actions from the equation and just make sure that our reducer works correct with the return values of those synchronous action creators.

Here is the function:

```
// source/store/actionUtils.js
export const asyncAction = ({ func, start, success, failure }) => (
  (...args) => (dispatch) => {
    dispatch(start());
    return func(...args)
      .then((data) => dispatch(success(data)))
      .catch((error) => dispatch(failure(error)));
  }
);
```

It takes a config object, with `func`, `start`, `success` and `failure` keys. As you might have already guessed: the last three will each contain the corresponding action creator. But what is `func`? It is a function our helper will call with all the arguments it receives. This function has to return a promise to which we will attach our `then` and `catch` handlers in order to dispatch the correct actions based on resolution/rejection.

The above helper returns a new function after receiving the config object, which is our action creator. But that action creator returns another function and not an action, so why is that? The returned function will be picked up by `redux-thunk` and gets called with two arguments: `dispatch` and `getState`. This allows us to do arbitrary complex, asynchronous logic in our action creator and dispatch actions at will. We are only using the `dispatch` argument here as our action does not require us to look at our state. But for the record, `getState` allows us to retrieve the store's state inside our action creator and use its data for our actions. This can be useful if we have to do a lot of processing or lookups, and do not want to funnel all of our data through our components.

Before starting our async request we will dispatch the action returned by `start()` to notify our application that our request is about to begin. This can be useful in order to show a loading indicator or disable a button until the action completes. After that we call the provided `func` to retrieve the promise. If the returned promise resolves, we will dispatch the result of `success()`. In case it gets rejected, `failure()` will be dispatched. We have to return the newly created promise from our action creator, in case some external consumer - like our tests - wants to wait for the promise to be resolved or rejected in order to process its result.

Now that we introduced all our helper functions let's see how quickly we are able to implement all our action creators!

FETCHING A LIST OF NOTES

As we previously did not have the need to fetch notes, this action creator is a new one. Using the above introduced helper functions, we can come up with the following test:

```
// source/store/actions.test.js
// imports and helpers omitted for brevity
test('action creator | fetchNotes ::', (t1) => {
  const NOTE = {
    id: 123,
    content: 'testing',
    timestamp: 456,
  };
  const func = () => actions.fetchNotes();
  const type = 'app/fetchNotes';
  const successPayload = { noteIds: [ NOTE.id ], notes: { [NOTE.id]:
NOTE } };
  const path = '/notes';
  const method = 'GET';
  const successMockReturn = [ NOTE ];
  const failureMockReturn = { body: {}, status: 400 };

  testAsyncAction(t1, {
    func,
    type,
    successPayload,
    path,
    method,
    successMockReturn,
    failureMockReturn,
  });
});
```

We only have to declare some variables now and pass them to our test helper via the config object. `type` is the type of the action we are expecting. `successPayload` is the payload we are expecting the success action to contain when it is being dispatched. `path`, `method`, `successMockReturn` and `failureMockReturn` are, as previously described, used to control what kind of data we mock for our `fetch` calls. Now this test covers the whole flow of our async action creator, we just need to make sure that we implement it as expected.

```
// source/store/actions.js
import api from '../utils/api';
import { startAction, successAction, failureAction, asyncAction } from './
actionUtils';

const fetchNotesType = 'app/fetchNotes';
export const fetchNotesStart = startAction(fetchNotesType);
export const fetchNotesSuccess = successAction(fetchNotesType);
export const fetchNotesFailure = failureAction(fetchNotesType);
export const fetchNotes = asyncAction({
  func: () => api.notes.fetch(),
  start: fetchNotesStart,
  success: fetchNotesSuccess,
  failure: fetchNotesFailure,
});
```

It only took us ten lines to implement the complete flow for fetching notes from our server, minus the API module and action helper code of course. Without those helpers, we would have had to write all of that code for each action.

ADDING A NOTE

Our previous action creator for adding a note was just a single plain function. As with `fetchNotes` above, we now have to write more extensive tests to cover the action creator's functionality:

```
// source/store/actions.test.js
// imports and helpers omitted for brevity
test('action creator | addNote ::', (t1) => {
  const successPayload = {
    id: 123,
    content: 'Hi',
    timestamp: 456,
  };
  const func = () => actions.addNote('Hi');
  const type = 'app/addNote';
  const path = '/notes';
  const method = 'POST';
  const successMockReturn = (url, { body }) => ({
    id: 123,
    content: JSON.parse(body).content,
    timestamp: 456,
  });
  const failureMockReturn = { body: {}, status: 400 };

  // call to `testAsyncAction` omitted for brevity
});
```

Again, we only have to change a few variables and we are good to go! This time we use a function to create the mock response in order to verify that the correct content is passed to the API method by the action creator.

That said, here is the code for our updated `addNote` action creator:

```
// source/store/actions.js
// imports and helpers omitted for brevity
const addNoteType = 'app/addNote';
export const addNoteStart = startAction(addNoteType);
export const addNoteSuccess = successAction(addNoteType);
export const addNoteFailure = failureAction(addNoteType);
export const addNote = asyncAction({
  func: (content) => api.notes.add(content),
  start: addNoteStart,
  success: addNoteSuccess,
  failure: addNoteFailure,
});
```

This code should look very very familar to you. Again our helpers allowed us to greatly reduce the amount of code we had to write. The only things we had to change were `type` and `func`.

UPDATING A NOTE

So far we have been just replacing our original action creators with the new, asynchronous ones. This does not work for updating a note, because we keep our note's local state in Redux and update it on every keystroke. Because of this, we are going to add a separate `updateNoteServer` action creator that we can later call explicitly using a save button. This action creator is supposed to call our API update method and return the new result:

```
// source/store/actions.test.js
// imports and helpers omitted for brevity
test('action creator | updateNoteServer ::', (t1) => {
  const successPayload = {
    id: 123,
    content: 'Hi',
    timestamp: 456,
  };
  const func = () => actions.updateNoteServer(123, 'Hi');
  const type = 'app/updateNoteServer';
  const path = '/notes/123';
  const method = 'PUT';
  const successMockReturn = (url, { body }) => ({
    id: 123,
    content: JSON.parse(body).content,
    timestamp: 456,
  });
  const failureMockReturn = { body: {}, status: 400 };

  // call to `testAsyncAction` omitted for brevity
});
```

Of course it is again easy for us to satisfy this test because of our helpers:

```
// source/store/actions.js
// imports and helpers omitted for brevity
const updateNoteServerType = 'app/updateNoteServer';
export const updateServerStart = startAction(updateNoteServerType);
export const updateServerSuccess = successAction(updateNoteServerType);
export const updateServerFailure = failureAction(updateNoteServerType);
export const updateNoteServer = asyncAction({
  func: (id, content) => api.notes.update(id, content),
  start: updateServerStart,
  success: updateServerSuccess,
  failure: updateServerFailure,
});
```

We only have one action to go!

REMOVING A NOTE

Last but not least, we need the action creator for removing a note. You will by now have sensed a pattern here and yes, our tests and the final action creator are going to resemble the previous ones, thanks to our useful helpers.

```
// source/store/actions.test.js
// imports and helpers omitted for brevity
test('action creator | removeNote ::', (t1) => {
  const func = () => actions.removeNote(123);
  const type = 'app/removeNote';
  const successPayload = { id: 123 };
  const path = '/notes/123';
  const method = 'DELETE';
  const successMockReturn = { status: 200, body: '' };
  const failureMockReturn = { body: {}, status: 400 };

  // call to `testAsyncAction` omitted for brevity
});
```

And just as before we only have to adjust a few variables to generate a correct test. And ten more lines will give us the code that will satisfy this test:

```
// source/store/actions.js
// imports and helpers omitted for brevity
const removeNoteType = 'app/removeNote';
export const removeNoteStart = startAction(removeNoteType);
export const removeNoteSuccess = successAction(removeNoteType);
export const removeNoteFailure = failureAction(removeNoteType);
export const removeNote = asyncAction({
  func: (id) => api.notes.delete(id).then(() => ({ id })),
  start: removeNoteStart,
  success: removeNoteSuccess,
  failure: removeNoteFailure,
});
```

A difference to our other action creators is that we are adding a function to the promise chain, which returns an object with the initial id. If you recall our API tests, you will remember that the delete method only returns a `200 OK` status, but no data. So, in order for our reducers to know which note has been deleted, we add a custom return value here.

We are now done implementing our action creators and can move on to update our reducers so they can work with the new actions!

Updating reducers

At the moment our reducers are not capable of understanding our asynchronous actions, and they will completely ignore the `meta` and `error` properties. They are also not able to handle our new `app/fetchNotes` action type. We are going to fix this right away!

Remember that we used a `getMockState` test utility before. We will add a new utility called `getActionPayload`, which will work just like `getMockState` but for action payloads. The code for this utility can be found in `source/store/testUtils.js`.

BYIDS **REDUCER**

As always, we first update our tests to reflect what we want our code to do.

`fetchNotesSuccess`

```
// source/store/reducers.test.js
// imports omitted for brevity
test('reducer | byId :: Handle "fetchNotesSuccess" action',
  ({ deepEqual, end }) => {
    const state = getMockState.withNoNotes();
    const fetchedNotes = getActionPayload.fetchNotesSuccess();
    const actualNextState = reducers.byId(
      state.byId,
      actions.fetchNotesSuccess(fetchedNotes)
    );
    const noteId = fetchedNotes.noteIds[0];
    const expectedNextState = {
      [noteId]: fetchedNotes.notes[noteId],
    };

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

Now it will become apparent why we previously separated our asynchronous actions from their synchronous counterparts. If we hadn't done that we would not be able to write such a straight forward test as we did above. This is because we'd have to get involved with promises, mocking `fetch` results, etc. Now we can just easily provide our success action with the expected data and test how our reducers react to it.

Making our test pass is easy:

```
// source/store/reducers.js
// imports omitted for brevity
export const byId = (state = {}, { type, payload, meta, error }) => {
  switch (type) {
    case 'app/fetchNotes':
      if (meta.done && !error) {
        return payload.notes;
      }
      return state;
    // rest of reducer omitted for brevity
  }
};
```

We have to inspect the `meta.done` and `error` properties to determine whether our action was successful. Only then do we want to return the new notes. As the `notes` property will already contain an object with all of the notes keyed by id, we do not need to do anything else than return it.

`addNoteSuccess`

Just as before we now have to use the synchronous `*Success` action in our test, instead of the previously used action, since that one is now asynchronous:

```
// source/store/reducers.test.js
// imports omitted for brevity
test('reducer | byId :: Handle "addNoteSuccess" action',
  ({ deepEqual, end }) => {
    const state = getMockState.withNoNotes();
    const newNote = getActionPayload.addNoteSuccess();
    const actualNextState = reducers.byId(
      state.byId,
      actions.addNoteSuccess(newNote)
    );
    const expectedNextState = {
      [newNote.id]: newNote,
    };

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

Making this test pass is no problem at all. We only have to look at the `meta.done` and `errors` props to make an informed decision whether to return the new or old state:

```
// source/store/reducers.js
// imports omitted for brevity
export const ids = (state = [], { type, payload, meta, error }) => {
  switch (type) {
    case 'app/addNote':
      if (meta.done && !error) {
        return prepend(payload.id, state);
      }
      return state;
    // rest of reducer omitted for brevity
  }
};
```

`updateServerSuccess`

When updating our note on the server, we expect our reducer to update itself properly with the result and overwrite any previous data for said note:

```
// source/store/reducers.test.js
// imports omitted for brevity
test('reducer | byId :: Handle "updateServerSuccess" action',
  ({ deepEqual, end }) => {
    const state = getMockState.withOneNote();
    const updatedNote = getActionPayload.updateServerSuccess();
    const actualNextState = reducers.byId(
      state.byId,
      actions.updateServerSuccess(updatedNote)
    );
    const expectedNextState = {
      [updatedNote.id]: updatedNote,
    };

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

Our implementation is exactly the same as for a normal note update, except for the fact that we have to wait until the action completes. We potentially could reuse our `updateNote` action and dispatch that when our server update succeeds, but using a separate action keeps things cleanly separted, even though it adds another case to handle.

```
// source/store/reducers.js
// imports omitted for brevity
export const byId = (state = {}, { type, payload, meta, error }) => {
  switch (type) {
    case 'app/updateNoteServer':
      if (meta.done && !error) {
        return merge(state, { [payload.id]: payload });
      }
      return state;
    // rest of reducer omitted for brevity
  }
};
```

`removeNoteSuccess`

The last action for this reducer is `removeNoteSuccess`. Let's create another failing test:

```
// source/store/reducers.test.js
// imports omitted for brevity
test('reducer | byId :: Handle "removeNoteSuccess" action',
  ({ deepEqual, end }) => {
    const state = getMockState.withOneNote();
    const actualNextState = reducers.byId(
      state.byId,
      actions.removeNoteSuccess({ id: 'id-123' })
    );
    const expectedNextState = {};

    deepEqual(actualNextState, expectedNextState);
    end();
  }
);
```

We are not using our `getActionPayload` utility here, since the payload we have to pass to `removeNoteSuccess` is small enough to not bloat our code.

Making the test pass is the same deal as before:

```
// source/store/reducers.js
// imports omitted for brevity
export const byId = (state = {}, { type, payload, meta, error }) => {
  switch (type) {
    case 'app/removeNote':
      if (meta.done && !error) {
        return dissoc(payload.id, state);
      }
      return state;
    // rest of reducer omitted for brevity
  }
};
```

`IDS` AND `OPENNOTEID` REDUCER

We have walked through the tests one-by-one for the `byId` reducer. We will not do so for the remaining two reducers since their updated tests are similar to the above and the gist is again that we have to inspect the `meta.done` and `error` properties. The updated code can be found in the Github repository.

Saving a note to the server

Right now we have no way of using our `updateNoteServer` action. Let's add a save button to our detail view so users can save notes whenever they feel like it!

Everything is ready so this will be straight forward. Since we want to be good citizens, we first add a new property to our `propTypes`:

```
// source/components/NoteDetail/index.jsx
NoteDetail.propTypes = {
  updateNoteServer: PropTypes.func.isRequired,
};
```

If you remember the `NoteDetail` view from the previous chapter, you know that our action creators were injected automatically. Now we just extract `updateNoteServer` from the props provided to our `NoteDetail` component and add a save button. When our user clicks the button we are simply going to call `updateNoteServer` with the note's id and content:

```
// source/components/NoteDetail/index.jsx
const NoteDetail = ({ note, removeNote, closeNote, updateNote, updateNote-
Server }) => (
  <div style={style.wrapper}>
    {!note
      ? <div style={style.blankslate}>No note is open</div>
      : <div style={style.note}>
          {/* previous elements omitted for brevity */}
          <div style={style.row}>
            <button
              onClick={() => updateNoteServer(note.id, note.content)}
              style={style.button}
              >
              Save
            </button>
            {/* next elements omitted for brevity */}
          </div>
        </div>
    }
  </div>
);
```

Whenever the save button is clicked, the note's new content will be persisted to the server.

Fetching notes on initial render

Fetching our notes couldn't be easier now. Just as before we add a new property to our `propTypes`:

```
// source/components/NotesList/index.jsx
NotesList.propTypes = {
  // rest omitted for brevity
  fetchNotes: PropTypes.func.isRequired,
};
```

Since all of our action and API code is ready, we only need to call `fetchNotes` in `componentWillMount` to kick off the request:

```
// source/components/NotesList/index.jsx
class NotesList extends Component {

  componentWillMount() {
    this.props.fetchNotes();
  }

  // rest omitted for brevity
}
```

That was quite easy, wasn't it? If you now start up your server and visit the application, it will not contain any notes. But when creating a new one and then refreshing the page, it will be loaded instantly. Perfect!

redux-thunk alternatives

As we've seen so far, `redux-thunk` is very powerful middleware to work with asynchronous actions. However, sometimes we may want to work with other constructs that adapt to our application's domain or our way of thinking better.

The interesting part of the whole middleware concept of Redux is that we can virtually do whatever we want with actions before they get to our reducers. The Redux community has gone very far in multiple directions allowing for many different ways to express logic through actions. In this section we'll explore two of them: using promises and defining our own custom middleware.

There's another approach many people are vouching for, called sagas. While the concept is outside of the book's scope, it is worth mentioning what it does and how you could benefit from it without going into much detail. In a nutshell, sagas work kind of like background processes that watch actions dispatched to the store and decide what to do based on them: they could make an asynchronous call to a service, dispatch other actions to the store, or even start other sagas dynamically. Sagas are built using generators and you achieve those tasks in the example above by yielding instructions to be executed, which are called effects. Despite it being a rather different way of structuring your application's logic, it might come in handy at some stage. Among its benefits, the library authors highlight its ease of testing and the ability to cancel actions at will. You can find more about it at the project's official repository: **https://github.com/yelouafi/redux-saga**.

THE PROMISE MIDDLEWARE

In the same way that a thunk middleware allows you to dispatch a function instead of an action, since middleware let's you dispatch a promise instead of an action.

Taken directly from the Redux docs, here's the simplest implementation of a promise middleware:

```
const vanillaPromise = store => next => action => {
  // if our action doesn't look like a promise, skip this middleware
  if (typeof action.then !== 'function') {
    return next(action);
  }

  // otherwise wait until that promise resolves and dispatch its result
  return action.then(store.dispatch);
}
```

For the sake of brevity, we won't go through the whole set of actions again, but we will focus on `addNote` instead.

Our current implementation is as follows:

```
const addNoteType = 'app/addNote';
export const addNote = asyncAction({
  func: (content) => api.notes.add(content),
  start: addNoteStart,
  success: addNoteSuccess,
  failure: addNoteFailure,
});
```

Let's take a step back from the lifecycle of our asynchronous action and assume for a moment that we only a successful call to our API will dispatch an `addNote` action. We will come back how to handle the lifecyle later.

```
export const addNote = (content) =>
  api.notes.add(content).then(payload => ({
    type: addNoteType,
    poyload
  }));
```

Since our API is already returning a promise, this middleware seems to be a natural fit for it.

When a user of this action dispatches it through the store, a reducer would ultimately see an action object like the following:

```
{
  "type": "app/addNote",
  "payload": {
    "id": "some-random-id",
    "content": "",
    "timestamp": 1468249934757
  }
}
```

That is a note ready to be added to our list. However, this would only happen if the server successfully replied to our request. So far, we have no way to indicate the reducer that either our action is in progress or that the server failed.

We could always dispatch an action before `addNote` and catch the error, if any are based on the promise that dispatch returned to us. However, that all sounds like a lot of extra and repetitive work, and we didn't sign-up to using the promise middleware to write more code, instead we want to use it because it helps us simplify our codebase.

What if our `addNote` action is as simple as returning an object; which payload would contain a promise that we could work on?

```
export const addNote = (content) => ({
  type: addNoteType,
  payload: api.notes.add(content)
});
```

In such a scenario, our promise middleware can check whether the payload is a promise, and if it is, it could provide the lifecyle logic we implemented in `asyncAction` above. Let's see how we ca make that happen!

```
const promiseMiddleware = store => next => action => {
  // check if we have a payload and if it is a promise
  if (!action.payload || typeof action.payload.then !== 'function') {
    return next(action);
  }

  // dispatch an action that tells the beginning of our action
  next({
    type: action.type,
    meta: {
      done: false
    }
  });
```

```
    // when the promise that the payload carries...
    return action.payload.then(
      // ...succeeds: dispatch an action with the result as the new payload
      result => next({
        type: action.type,
        payload: result,
        meta: {
          done: true
        }
      }),
      // ...if it fails: dispatch an action with the error as the payload
      error => next({
        type: action.type,
        payload: error,
        error: true,
        meta: {
          done: true
        }
      })
    );
  }
```

Notice how our new middleware makes use of the `meta` and `error` keys in the actions it dispatches to represent the state of our request in a similar way that our `asyncAction` method would have done before. The difference now relies on the fact that our actions have become much easier to follow and reason about because they are just plain objects. An interesting side effect of using promises is that a component that dispatched such an action could wait for it to resolve to do something else that is linked to that action.

If you plan on using this middleware in production, please note that it makes a simplification and it discards anything other than the type and payload that your action might be carrying. If you have extra fields that you would like your reducer to receive, you will want to append those to the dispatched actions.

Also, please note that while the middleware we've created here works well, in practice you may want to choose one maintained by the community so you don't have to do it yourself and write tests for it. Here are some packages worth looking at that implement this pattern:

- **redux-promise** **https://github.com/acdlite/redux-promise**: it doesn't support optimistic updates on the stable version.
- **redux-promise#sequence** **https://github.com/acdlite/redux-promise/tree/sequence**: the same library has a branch called `sequence` that supports the pattern implemented above. It has been released to npm under version `0.6.0-alpha`, but because people haven't settled on the approach to these kind of updates, it hasn't been merged yet. Having said that, it is tested and works very well. You can track the issue here: **https://github.com/acdlite/redux-promise/issues/12**.

- **redux-promise-middleware https://github.com/pburtchaell/redux-promise-middleware**: it follows a different approach and suffixes actions instead of using `meta`.

Testing our promise middleware powered action

With this test exercise we'll take the opportunity to introduce a more isolated approach to testing through `proxyquire`, which allows us to swap dependencies in modules for whatever we want. We will leverage that when testing our action to replace our dependency on the API calls, so we don't have to mock fetch at all. If you're interested on it, you can dive deeper on `proxyquire` at the project's page on GitHub: **https://github.com/thlorenz/proxyquire**.

Our actions do much less now: we only need to test that it carries a promise in its payload and has the right type. This is what our test could look like:

```
import proxyquire from 'proxyquire';
import test from 'tape';

test('action creator | addNote ::', ({ end, equals }) => {
  // proxyquire let's you swap dependencies for whatever you define.
  // in our example, actions.js calls:
  // import api from '../utils/api';
  // if we don't care about api for our test, we can simply swap it for what-
ever object we want.
  // In our example we're replacing addNote for a method that when called
will only return
  const { addNote } = proxyquire('./actions', {
    '../utils/api': {
      addNote: content => Promise.resolve({id: 'new', content, timestamp:
'now'}),
      // ...
    }
  });

  const content = 'content';
  const type = 'app/addNote';

  // call our action creator
  const action = addNote(content);

  // ensure we get the right type
  equals(action.type, type, 'sets the right type');
  // ensure we get a promise
  equals(typeof action.payload.then, 'function', 'the payload is a Promise');

  action.payload.then(result => {
    // check that our promise resolves with the expected result
    equals(result.content, content, 'calls api.addNote and adds a note with
the content provided');
```

```
    end();
  });
});
```

Combining promises with thunks

An important thing to remember is that middleware runs one after the other and that wherever we have access to `dispatch`, such as from a thunk, we can dispatch any other type of action and it will run through the whole chain again.

Let’s say that a new requirement came in and our notes app now takes a title and it requires that each title is unique. One possible solution to this problem would be to get a list of all titles into our React components and, before dispatching the action, check that the title doesn’t already exist. That would require our component to get more data than it needs. It would also need to get the list of titles for all notes. However, the note component should be concerned with its own note and nothing else.

How do we manage to keep it simple and still deliver this feature? We use a thunk! Remember that thunks get two parameters: `dispatch` and `getState`. Such an action creator could look like this:

```
export const maybeAddNote = (title, content) => (dispatch, getState) => {
  // getTitlesFromNotes would be a selector to get titles from all the notes
we have on state
  const titles = getTitlesFromNotes(getState());

  if (!titles.includes(title)) {
    dispatch(addNote(title, content));
  }
}
```

CUSTOM API MIDDLEWARE

If the status quo of dispatching actions doesn’t express your needs very well, you can always create your own middleware that does exactly what you want!

In this section we’ll briefly explore how we can get an action like this:

```
export const addNote = (content) => ({
  type: addNoteType,
  payload: content,
  api: {
    name: 'notes',
    method: 'addNote'
  },
});
```

This works with our API without explicitly calling any methods in our action, and instead describes what we want it to do.

This is what our middleware could look like:

```
// we assume we have a central point that deals with all of our APIs and
//
// such a file can export * as notes from './notes';
//
// making our import give an object that holds the notes API
import * as apis from './apis';

const apiMiddleware = store => next => action => {
  // if the action doesn't involve any API stuff, move on
  if (!action.api) {
    return next(action);
  }

  // get the API
  const api = apis[actions.api.name];
  // get the method
  const method = api[actions.api.method];

  // call our API method and dispatch the action in case of success
  method(action.payload)
    .then(result => next({
      type: action.type,
      payload: result
    }));
};
```

For simplicity's sake, this implementation avoids checks to make sure that both the API and the method we're trying to call method exist. It also doesn't provide any sort of lifecycle actions around the current one, but that could be easily introduced in a similar way than that of previous sections of this chapter.

The key within our action is the `api` property (you can choose any name for it, it could even be a value that your API only understands). Our custom middleware then uses that information to map it out to methods we have already defined and that's about it. This is a very simple example and it hopefully shows the potential it has. Its behavior can be as simple and as complex as you want.

An interesting side effect of this approach is that testing your action creator is super simple:

```
test('custom api action', ({ deepEquals, end }) => {
  const content = 'content';

  deepEquals(
    addNote(content),
    {
      type: 'notes/addNote',
      payload: content,
      api: {
        name: 'notes',
        method: 'addNote'
      }
    }
  );

  end();
});
```

This approach opens an interesting possibility. Since you're not calling your API directly from within your action, you could very easily swap it at the middleware layer based on whatever criteria you define.

Conclusion

We have made a lot of progress on our sample app. By taking advantage of the `redux-thunk` middleware, our app now has server-side persistence. In the next chapter we will add the feature of communicating network request states to our sample app.

Showing Request State to Users 6

In the previous chapter, we built the base functionality of syncing our applications data with the server. This makes our application significantly more useful than it was before, because users can now access their data across multiple devices. But we are far from done! Getting the app to *perform* asynchronous requests is not the same as actually *communicating* this asynchronicity to the user. Network requests take time and don't always succeed.

Modern applications go to great lengths to make sure that the user always feels in control; communicating network request states is essential to achieving this. In this chapter, we will look into how to achieve this with Redux.

Note: You can find the complete source code for the sample application at **https://github.com/arturmuller/developing-a-redux-edge**.

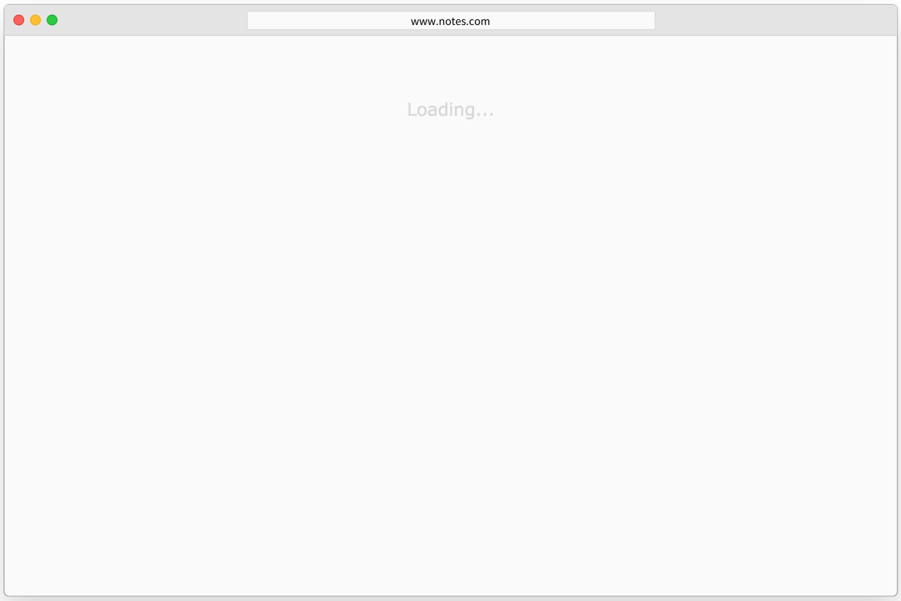
App spec

There are several changes that we want to introduce to the app in order to keep the user in the loop at all times.

Pending, success, and failure states

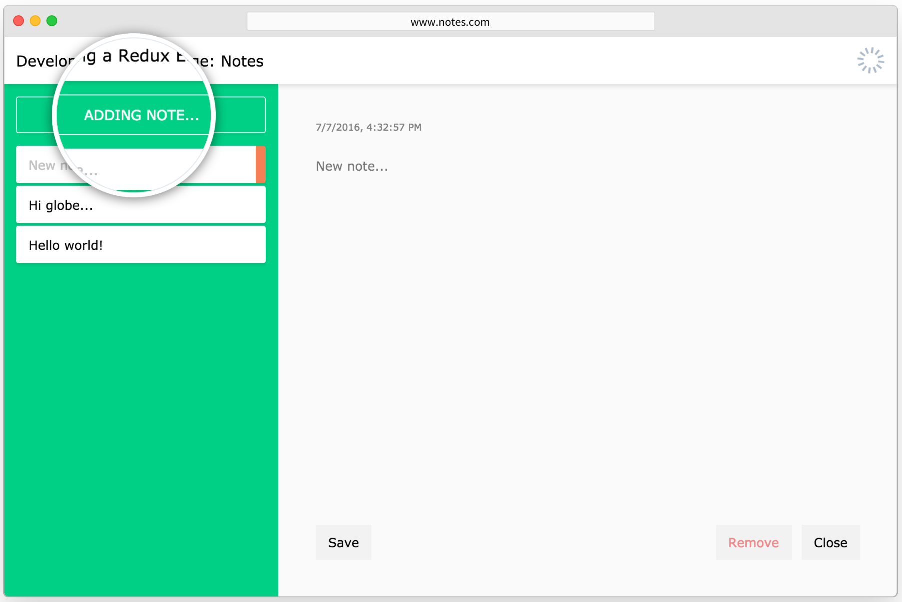
Any component that fires off an async action should communicate that the action has been triggered and that we're now waiting for a response.

The read notes list request is triggered automatically when the app component mounts, so we want to show a loading indicator there immediately, and keep it there until the request is fulfilled.

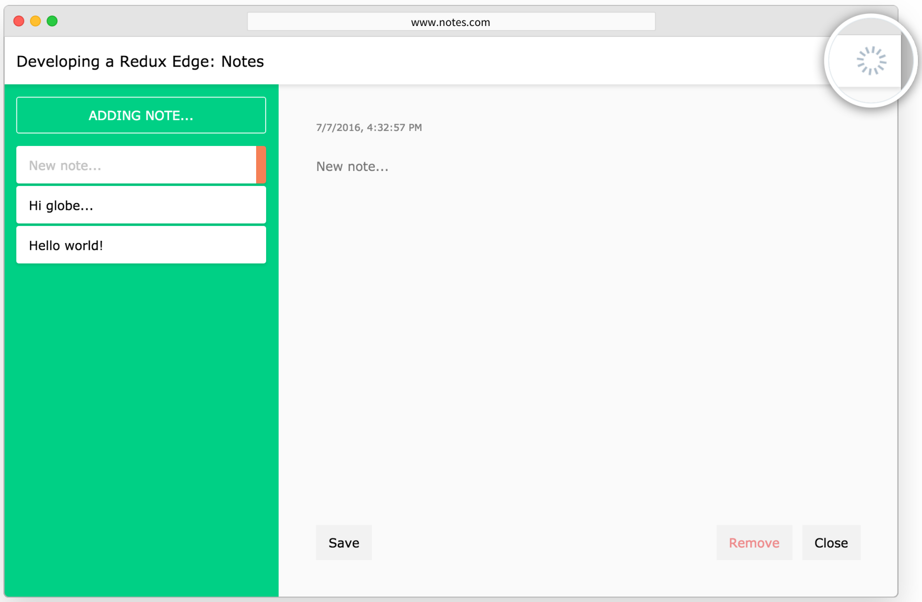


If the request fails, we will show any potential failures directly inside the view as well.

For create, update, and delete note requests, we want to show the pending state on the button itself by changing the label of the button when the action is in progress. ‘Save’ becomes ‘Saving...’, ‘Remove’ becomes ‘Removing...’, and so on. We also want to disable the button when a request is pending to prevent users accidentally firing off multiple actions.

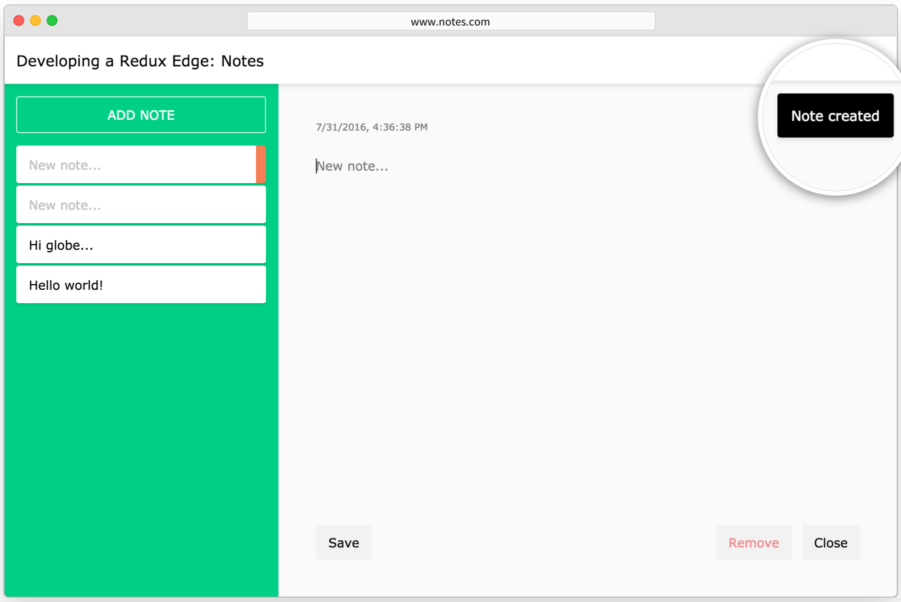


Lastly, we want to have a network status indicator in the top right corner of the app. This indicator will start spinning whenever any network request is pending.



Toast notifications

For the read notes list, the success state is pretty obvious; our freshly received entities appear inside the UI. On the other hand, for create, update and delete note actions, the fact that the request has succeeded (or failed) is not always so obvious. For that reason, we'll show a toast notification upon success or failure depending on the request response.



Obviously, we don't want these notifications hanging around forever. Any toast should be visible for a maximum of 3 seconds, unless it is replaced with a newer one. We only ever want to show one toast at a time.

Implementation

Now that we have our brief, let's jump into actually building the above features!

Logger

First, we'll install Redux Logger. This is done so that we are not building our features blind. Up until now, actions happened one at a time and were almost always triggered directly by the user. That won't be the case by the end of this chapter, so being able to see what's happening (without having to console.log everything!) will be invaluable.

You should `npm install -S redux-logger` and then add the following to your `store/index.js` file:

```
// source/store/index.js

import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import createLogger from 'redux-logger';
import rootReducer from './reducers';

const logger = createLogger({ collapsed: true });

export default function configureStore() {
  const store = createStore(rootReducer, applyMiddleware(thunk, logger));
  return store;
}
```

Now that we have added Redux logger to our middleware stack, if you open up your browser's console, you'll see every action that passes through Redux.

```
action @ 10:58:56.416 app/replaceNotes
action @ 10:59:11.325 app/openNote
action @ 10:59:18.231 app/updateNote
```

If you're using Chrome, you will be able to expand each action and inspect it's contents, as well as the before and after state. (Other browsers might just leave everything expanded all of the time.)

Requests

To be able to access the state of any network request the app makes — from any component — we will need to store it inside the main Redux state.

For each request, we are interested in it's status — pending, success, failure — and, in the case that it fails, we also want to keep the information about the error around.

The state slice for each request should end up looking like this:

```
// Initial state
{
  status: 'pending',
  error: null,
}

// Success state
{
  status: 'success',
  error: null,
}

// Failure state
{
  status: 'failure',
  error: {
    message: 'Not Found',
  },
}
```

But where inside the state can we put these objects? Let's look at how our state is looking at the moment:

```
// Current state shape
{
  byId: {},
  ids: [],
  openNoteId: null,
}
```

A naive approach would be to simply add a new state slice for every request we currently have in the app manually. This would end up looking something like this:

```
// Proposed state shape #1
{
  byId: {},
  ids: [],
  openNoteId: null,
  requestReadNotes: { ... },
  requestCreateNote: { ... },
  requestDeleteNote: { ... },
  requestUpdateNote: { ... },
}
```

Unfortunately, this approach is really not that great.

First, if you imagine you would have to write separate reducers and action creators for each of these state slices, it would really add some bloat to the codebase. Of course, you *could* write some helper factories to generate said reducers and action creators programatically, but that's still probably not the best idea. Helpers almost always add a level of opaqueness to your code (so they should be used with caution), and generating each request type one-by-one is actually still a lot of manual and error prone work.

Second, this state shape doesn't account for the fact that delete note and update note are dynamic.

Unlike requests to read and create, which always have the same URL (`/api/notes/`), delete and update target a specific note (using its id) in the URL: `/api/notes/${id}`. It would be incorrect to only keep one status for all of the possible ids. If we did that, the user could, for example, click 'Remove' on one note, navigate to a different note, and, if the request was taking a while to come back, still see the 'Removing...' label on the new note. Surely that would give them quite a fright!

For this reason, we need a reducer that can store an arbitrary number of requests under dynamically generated keys.

We want it to like this:

```
// Proposed state shape #2
{
  byId: {},
  ids: [],
  openNoteId: null,
  requests: {
    "readNotes": { ... },
    "createNote": { ... },
    "updateNote/a12b75ad-e06b-41f3-abbb-f6dcfae5c969": { ... },
    "updateNote/5c500147-9525-4269-bd9d-d04ad44b21b3": { ... },
    "deleteNote/a236c29f-2685-4ab4-b181-e58c10c45550": { ... },
  }
}
```

Notice how we are using the requests IDs as part of the property name to differentiate between requests like `updateNote`.

Now, the question is how do we generate the key? Well, the most flexible way to do this is to simply leave it up to each request to register itself under whatever key makes sense for it. Whomever is writing the async action will know best how to create a unique key from what they currently have at their disposal. (Just like you might do with React element keys).

Nevertheless, it is a good idea to devise a consistent scheme because we will need to use it in our selectors to retrieve the request. In our case, we will concatenate the API method name with any arguments passed to the method, separated by a forward slash (as seen above).

Now that we have a good idea of what we're trying to build, let's actually do so. We'll start with the synchronous part of the process:

```
// source/store/actions.js
//...

export const markRequestPending = (key) => ({
  type: 'app/markRequestPending',
  meta: { key },
});

export const markRequestSuccess = (key) => ({
  type: 'app/markRequestSuccess',
  meta: { key },
});

export const markRequestFailed = (error, key) => ({
  type: 'app/markRequestFailed',
  payload: { error },
  meta: { key },
});
```

```
// source/store/reducers.js
// ...

export const requests = (state = {}, { type, payload, meta }) => {
  switch (type) {
    case 'app/markRequestPending':
      return merge(state, { [meta.key]: { status: 'pending', error:
null } });
    case 'app/markRequestSuccess':
      return merge(state, { [meta.key]: { status: 'success', error:
null } });
    case 'app/markRequestFailed':
      return merge(state, { [meta.key]: { status: 'failure', error: pay-
load.error } });
    default:
      return state;
  }
};

export default combineReducers({
  byId,
  ids,
  openNoteId,
  requests,
});
```

Let's quickly recap what's happening here:

Action creators return objects as usual, so nothing new here.

The `requests` reducer grabs a `key` from `meta`, and depending on the action, populates it with the correct data. Finally, the reducer itself is then added to the root reducer.

You might be wondering why we use `meta` instead of dumping everything into payload. There is no hard and fast rule for when to use `meta` and when to use `payload` in the FSA spec. Generally, though, you'd often see it used to pass extra data to reducers that is to be used by the state itself or to provide data to middleware with which side-effects might be executed. In this case, `meta.key` will be used to generate the property name inside the state, but it won't actually be something our selectors will target. You won't ever be selecting the request key.

One more tip about using `meta` and, in fact, `payload` too: Most of the time it is handy to use an object to 'name' the contents of these generic keys. Doing so will help you prevent ambiguous code like the below:

```
export const reducer = (state = {}, { type, payload }) => {
  switch (type) {
    case 'action1':
      return merge(state, { [payload]: "foo" });
    case 'action2':
      return merge(state, payload);
    default:
      return state;
  }
};
```

What is `payload`? We can't know unless we look at our action creators. At best we can guess their type: in the case of `action1` it is either a string or a number and for `action2` it should be an object (because we know that Ramda's `merge` accepts two objects). But we still don't know either of these payloads' meaning.

Consider this much more explicit approach:

```
export const reducer = (state = {}, { type, payload }) => {
  switch (type) {
    case 'action1':
      return merge(state, { [payload.id]: "foo" });
    case 'action2':
      return merge(state, payload.note);
    default:
      return state;
  }
};
```

Here, payload is *always* an object. For `action1` we see that it carries an *id* and `action2` deals with the entire *note*. This is much clearer, and much more maintainable.

And that concludes our sync stuff. Now to actually use the `markRequest...` actions. Let's explore how we want to do that with the `requestUpdateNote` thunk.

Currently we have:

```
// source/store/actions.js
// ...

export const requestUpdateNote = (id, content) => (dispatch) =>
  api.notes.update(id, content).then((note) => dispatch(updateNote(note)));

// ...
```

What we need to do is dispatch `markRequestPending` just as the action starts, and then, depending on how it is resolved, we either `markRequestSuccess` or `markRequest-Failed`. We also must not forget to fire off the primary success action! In this case `update-Note`.

```
// source/store/actions.js
// ...

export const requestUpdateNote = (id, content) => (dispatch) => {
  const key = `updateNote/${id}`;
```

```
    dispatch(markRequestPending(key));
    return api.notes.update(id, content)
      .then((notes) => {
        dispatch(updateNote(notes));
        dispatch(markRequestSuccess(key));
      })
      .catch((reason) => {
        dispatch(markRequestFailed(reason, key));
      });
  }

  // ...
```

This works, but surely you can imagine that if you write this kind of code for every request, you will have to write a *lot* of code. It is fragile, not very testable, and it also definitely *obscures intent*. If you want to understand what the above actually does, you have to mentally filter out all of the `dispatch` calls, the chained `then`/`catch` methods. We can do better.

We want to be able to look at our code and immediately see what actions will be fired at the start, which on success, and which on failure. We also want to make sure we don't ever forget to fire off the `markRequest...` actions. In fact, we'd like these to be considered an implementation detail, and we don't really want to worry about them on request by request basis.

We'll need to abstract the above into a generic request action helper. We want to achieve the following:

1. Allow us to specify an arbitrary number of actions to be dispatched at one of the three points in the request's lifecycle.
2. Make sure that `markRequestPending`, `markRequestSuccess`, and `markRequest-Failed` are always fired off the the right points.

Let's start by generalizing the `requestUpdateNote` thunk into a `createRequestThunk` helper.

```
export const createRequestThunk =
  ({ request, start = [], success = [], failure = [] }) => {

  return (...args) => (dispatch) => {
    start.forEach((actionCreator) => dispatch(actionCreator()));

    return request(...args)
      .then((data) =>
        success.forEach((actionCreator) =>
          dispatch(actionCreator(data))))
      .catch((reason) =>
        failure.forEach((actionCreator) =>
          dispatch(actionCreator(reason))));
  };
};
```

This is not too different from the literal implementation. The main difference being that instead of dispatching actions directly, we are iterating over `start`, `success` and `failure` arrays with `forEach` and dispatching actions within. Note the use of default arguments to make sure we have an array in case none is supplied.

Also, notice how all arguments passed to the thunk are moved to the `request` using `...args`.

Lastly, every `actionCreator` inside success and failure hooks is passed the data returned from the promise. They wouldn't be very useful otherwise!

Ok, now to tackling automatically dispatching `markRequest...` actions. We need to do two additional things:

1. Add the correct `markRequest...` action creator at the end of each supplied action creators stack.
2. Generate the request key and pass it to these action creators.

Our final implementation will look like this:

```
export const createRequestThunk =
  ({ request, key, start = [], success = [], failure = [] }) => {

  return (...args) => (dispatch) => {
    const requestKey = (typeof key === 'function') ? key(...args) : key;

    start.forEach((actionCreator) => dispatch(actionCreator()));
    dispatch(markRequestPending(requestKey));
    return request(...args)
      .then((data) => {
        success.forEach((actionCreator) => dispatch(actionCreator(data)));
        dispatch(markRequestSuccess(requestKey));
      })
      .catch((reason) => {
        failure.forEach((actionCreator) => dispatch(actionCreator(reason)));
        dispatch(markRequestFailed(reason, requestKey));
      });
  };
};
```

The interesting part here is how we generate our `requestKey`. You'll notice that if it is a function, we call it with the arguments supplied to the thunk, otherwise we just assign it to `requestKey` directly.

This way, we handle both the case in which we have a 'static' request like `requestReadNotes`, where we simply want to provide the string literal — `readNotes` — and in the case in which 'dynamic' requests like `requestUpdateNote`, need to construct the key from the arguments passed to it.

Great. Using our fresh-off-the-press `createRequestThunk` inside the app is looking good:

```
// source/store/actions.js
//...

export const requestReadNotes = createRequestThunk({
  request: api.notes.readList,
  key: 'readNotes',
  success: [
    replaceNotes,
    (notes) => (notes.ids.length > 0)
      ? openNote(notes.ids[0])
      : noop('No note to open'),
  ],
});

export const requestCreateNote = createRequestThunk({
  request: api.notes.create,
  key: 'createNote',
  success: [ insertNote, (note) => openNote(note.id) ],
});

export const requestUpdateNote = createRequestThunk({
  request: api.notes.update,
  key: (id) => `updateNote/${id}`,
  success: [ updateNote ],
});

export const requestDeleteNote = createRequestThunk({
  request: api.notes.delete,
  key: (id) => `deleteNote/${id}`,
  success: [ (note) => removeNote(note.id) ],
});
```

Here are a couple of things to note:

We are explicitly dispatching `openNote` after `insertNote` as a reaction to `requestCreateNote` and `requestReadNotes` success.

In our previous implementation, the `insertNote` action was picked up by the `byId` and `ids` reducers, as well as the `openNoteId` reducer, which just happened to set the `openNoteId` to the correct value.

Sharing actions across reducers is OK if their effects are directly related (like with `byId` and `ids`), but in the case of `openNoteId`, it is hard to justify.

You have to have intimate knowledge of your reducers to understand why a note opens after it has been inserted. If you simply read the action name, it would not be immediately obvious. It will also not be visible in the log. Since you already *have* an `openNote` action, you might be pretty surprised that a note has opened even though this action was not dispatched.

The fact that `insertNote` is also doing the job of `openNote` also means that these two effects are now intimately coupled. If you want to reuse the `insertNote` action for purposes where it *doesn't* open a note, you cannot. As you can probably imagine, inserting a note into the state is not an uncommon action. You might want to do it as a reaction to various requests, incoming websocket events, etc.

For these reasons, it is best to keep your actions as small and decoupled as possible, and use ‘action sharing’ across reducers cautiously.

You might have also noticed how inside the `requestReadNotes` success stack, we open the note conditionally, only if there are notes present in the response. But because every entry in the stack will be passed to `dispatch`, we need to return *something*.

Although we could update the `createRequestThunk` helper to recognize invalid values and not dispatch them, returning a noop (No Operation) action with an explanation is a handy way to make it explicit that we didn't want anything to happen.

This is cool when rewriting/refactoring this very code, but also when looking at the actions log. It clearly separates bugs (something was supposed to dispatch, but didn't) from intended lack of reaction.

`noop` looks like this:

```
// source/store/actions.js
// ...

export const noop = (explanation) => ({
  type: 'app/noop',
  payload: explanation,
});

// ...
```

For completion, we should also now cleanup `openNoteId` to just listen to the two actions that *should* concern it: `openNote` and `closeNote`.

That wraps up our actions. Now is the time to prepare selectors.

If we look back at our brief, we will notice that we need two different kinds of data from the state.

1. For buttons and views, we need the individual request objects (such as `{ status: 'pending', error: null }`) so that we can render the correct version of these components.

2. For the global request indicator, we need to know whether *any* request is currently pending. For that we will simply need a `true`/`false` flag.

```
// source/store/selectors.js

export const getRequest = (state, key) =>
  state.requests[key] || {};

export const getRequests = (state) =>
  state.requests;

export const areRequestsPending = (requests) => {
  return Object.keys(requests)
    .some((key) => requests[key].status === 'pending');
};
```

`getRequest` is pretty straight forward, except you might be wondering why do we return an empty object when no request is found? The short answer is: for convenience. We will explore why exactly once we get to writing our components. Right now, the takeaway is that it is okay to occasionally return 'fake' state from selectors. Keep in mind that this is definitely something to be used with caution and best reserved for cases where you want an alternative way of saying 'entity not found.'

`getRequests` and `areRequestsPending` are again designed to be used in tandem inside components' `mapStateToProps`, just like we have done with `getNote`.

We will do that right now, inside a new component called Nav. This is where our global network request indicator will live.

```
// source/components/Nav/index.jsx

import React from 'react';
import { connect } from 'react-redux';
import * as selectors from '../../store/selectors';
import * as styles from './styles';

const Spinner = () => (
  <svg style={styles.spinner} viewBox="0 0 42 42" width="2.8em"
height="2.8em">
    <g stroke="none" fill="none" transform="translate(1, 1)">
      // ...SVG paths markup omitted for brevity.
    </g>
  </svg>
);

const Nav = ({ areRequestsPending }) => (
  <div style={styles.wrapper}>
    <h1 style={styles.heading}>Developing a Redux Edge: Notes</h1>
    {areRequestsPending && <Spinner />}
  </div>
);

const mapStateToProps = (state) => ({
  areRequestsPending: selectors.areRequestsPending(
    selectors.getRequests(state)),
});

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

As promised, `areRequestsPending` and `getRequests` are composed to provide the final data.

We have Nav, but currently it is not rendered anywhere.

Let's update app to use the Nav component and to use the `readNotesRequest` object to render pending and error states.

```
// source/components/App/index.jsx
// ...

class App extends Component {
  componentWillMount() {
    this.props.requestReadNotes();
  }

  render() {
    const { readNotesRequest } = this.props;

    switch (readNotesRequest.status) {
      case 'success':
        return (
          <div style={style.wrapper}>
            <Nav />
            <div style={style.row}>
              <NotesList />
              <NoteDetail />
            </div>
          </div>
        );
      case 'failure':
        return (
          <div style={style.notice}>
            {(readNotesRequest.error.message === 'Failed to fetch')
              ? 'No connection, try again later!'
              : 'Hmm... Something didn\'t go as planned.'
            }
          </div>
        );
      default:
        return (
          <div style={style.notice}>
            Loading...
          </div>
        );
    }
  }
}

const mapStateToProps = (state) => ({
  readNotesRequest: selectors.getRequest(state, 'readNotes'),
});

export default connect(mapStateToProps, actionCreators)(App);
```

The interesting part here is the new `switch` statement. We use `status` on `readNotesRequest` to decide which version of the component to render.

Notice how handy the empty object is here. If there was a possibility of getting undefined from `readNotesRequest`, we'd have to perform an additional check before accessing `status` and we'd have to do this on every component that uses a request. Not fun.

Also notice how we fall through to the pending state. This is done so that if this component starts rendering before the request was started (and the selector returns `{}`) we still end up rendering "Loading...".

In the `failure` case, we are using `readNotesRequest.error` to handle different errors differently. In our case we are differentiating between only two cases (no connection and any other error), but it is not hard to imagine using another switch statement to achieve a more fine grained control over our error handling.

Ok, all that is remaining to complete our brief for request state is making sure our buttons show the present continuous version of the label and are disabled during the time that the request is ongoing.

This process is exactly the same for all cases so we'll only cover it on once to avoid unnecessary repetition. We'll do so on the 'Save' button inside NoteDetail component.

As usual, the first thing we have to do is pass state to our components using `mapStateToProps` and our `selectors`.

```
// source/components/NoteDetail/index.jsx
// ...

const mapStateToProps = (state) => {
  const openNoteId = selectors.getOpenNoteId(state);
  return {
    note: selectors.getNote(state, openNoteId),
    updateNoteRequest: selectors
      .getRequest(state, `updateNote/${openNoteId}`),
    deleteNoteRequest: selectors
      .getRequest(state, `deleteNote/${openNoteId}`),
    };
  };

  // ...
```

`openNoteId` is used here to try to select requests relevant to this particular note.

Now, assuming we have received `updateNoteRequest` in props, we can use the request object in our components.

```
// source/components/NoteDetail/index.jsx
// ...

<button
  onClick={() => requestUpdateNote(note.id, note.content)}
  disabled={updateNoteRequest.status === 'pending'}
  style={style.button}
  >
  {updateNoteRequest.status === 'pending'
    ? 'Saving...' : 'Save'}
</button>

//...
```

This does the job, but it feels pretty messy and brittle. If we create a new `Button` component, we can really clean this up:

```
// source/components/NoteDetail/index.jsx
// ...

<Button
  label='Save'
  labelPending='Saving...'
  onClick={() => requestUpdateNote(note.id, note.content)}
  pending={updateNoteRequest.status === 'pending'}
  />

//...
```

The implementation of `Button` can be found on GitHub if you're interested.

One thing you might be thinking is: why not actually `connect()` the `Button` component as well? We could avoid manually selecting the request states from the store and instead just specify the requestKey on the `Button`, which would then handle the selection on it's own.

This is certainly possible, and it might be a good idea depending on your use-case. Nevertheless, there are also advantages to keeping your leaf components. That is, components that are more like ‘elements', such as Buttons, Toggles, Inputs, etc. should be as simple as possible.

In React/Redux circles, components that simply render markup given some props are often called dumb components. Dumb components are awesome. They are extremely reusable, composeable, and robust.

Your application will inevitably change over time, and the fewer components know about the details of it, the easier it is to adjust your codebase.

Dumb components can also be very easily packaged as modules and shared across codebases without enforcing a specific state management library or state shape. Lastly, dumb components are just easier to reason about and write.

Because of the above, we will leave our Button blissfully ignorant of the surrounding world.

Note that keeping Button dumb means that it can be used for non-async cases too:

```
<Button
  label='Close'
  onClick={closeNote}
  margin='left'
  />
```

To finish off our request’s state feature, we just replace all buttons with `Buttons`, inside both `NoteDetail` and `NotesList`, and we are done!

Toasts

Now that we have successfully completed requests, it is time to look at the second part of the puzzle: how to deal with the toast notifications.

Just to recap: we want to use toasts to communicate success and failure in the situation that it is not immediately obvious from the UI. For our particular use-case, the simplest way to tackle this is to simply keep the toast object inside the state.

The base is pretty standard stuff that we have already seen several times, so let’s just breeze through it:

```
// source/store/selectors.js
// ...

export const setToast = (message, level = 'info', id = v4() ) => ({
  type: 'app/setToast',
  payload: {
    id,
    message,
    level,
  },
});
```

```
export const clearToast = (timestamp) => ({
  type: 'app/clearToast',
  payload: { timestamp },
});

// ...

export const requestCreateNote = createRequestThunk({
  request: api.notes.create,
  key: 'createNote',
  success: [
    insertNote,
    (note) => openNote(note.id),
    () => setToast('Note created')
  ],
  failure: [
    () => setToast('Couldn\'t add note', 'warn'),
  ],
});

export const requestUpdateNote = createRequestThunk({
  request: api.notes.update,
  key: (id) => `updateNote/${id}`,
  success: [
    updateNote,
    () => setToast('Note saved'),
  ],
  failure: [
    () => setToast('Couldn\'t save note', 'warn'),
  ],
});

export const requestDeleteNote = createRequestThunk({
  request: api.notes.delete,
  key: (id) => `deleteNote/${id}`,
  success: [
    (note) => removeNote(note.id),
     () => setToast('Note deleted'),
  ],
  failure: [
    () => setToast('Couldn\'t remove note', 'warn'),
  ],
});
```

And here is reducers.js:

```
// source/store/reducers.js
// ...
```

```
export const toast = (state = null, { type, payload }) => {
  switch (type) {
    case 'app/setToast':
      return payload;
    case 'app/clearToast':
      return null;
    default:
      return state;
  }
};

// ...

export default combineReducers({
  byId,
  ids,
  openNoteId,
  toast,
  requests,
});
```

And finally, selectors.js:

```
// source/store/selectors.js
// ...

export const getToast = (state) =>
  state.ui.toast;

// ...
```

This is what we have done:

1. Built our `setToast` and `clearToast` action creators.
2. Placed them inside our `requestThunks`' success/failure stacks to be triggered at the appropriate times.
3. Built the corresponding reducer and added it to our root reducer.
4. Prepared the only necessary selector.

The only remaining feature for toasts is dismissing them after 3 seconds.
One way to do this is to write a thunk. This solution could look something like this:

```
// Proposed solution for auto-dismissing toasts #1

export const flashToast = (...args) => (dispatch) => {
  dispatch(setToast(...args));
```

```
    window.setTimeout(() => dispatch(clearToast()), 3000);
  };
```

This definitely works, but it also hides one subtle bug. If a second toast fires off before the first one one is dismissed by the thunk, it will not be dismissed in 3 seconds, but in whatever was the reminder of the time for the original thunk.

Obviously, this is not ideal.

There are various ways to solve this: we could `clearToast` by id instead of just clearing *any* toast, or we could `clearTimout()` for the original toast if we receive a new thunk (but that would require keeping the timeout id around somewhere).

But in this case, the most correct way of solving this is to not use a thunk at all. Everything we want to do can be achieved from the Toast component itself:

```
// source/components/Toasts/index.jsx

import React from 'react';
import { connect } from 'react-redux';
import * as actionCreators from '../../store/actions';
import * as selectors from '../../store/selectors';
import * as styles from './styles';

class Toast extends React.Component {
  componentDidMount() {
    this.timeout = setTimeout(
      () => this.props.clearToast(this.props.id),
      3000
    );
  }
  componentWillUnmount() {
    if (this.timeout) {
      clearTimeout(this.timeout);
    }
  }
  render() {
    const { id, message, level } = this.props;
    return (
      <div key={id} style={{ ...styles.toast, ...styles[level] }}>
        {message}
      </div>
    );
  }
}

const Toasts = ({ toast, clearToast }) => (
  <div style={styles.wrapper}>
    {toast &&
      <Toast
        {...toast}
```

```
          key={toast.id}
          clearToast={clearToast}
          />
      }
    </div>
  );

const mapStateToProps = (state) => ({
  toast: selectors.getToast(state),
});

export default connect(mapStateToProps, actionCreators)(Toasts);
```

As you can see, the timeout is set (and kept) on the instance of the Toast component. Each Toast manages its timeout on its own.

Because we're using the Toast's ID for the `key` prop when we're using it, React will re-render the entire component if the id changes. The currently active toast will unmount, clean up after itself, and new Toast will mount, setting a fresh timeout.

As you can imagine, handling multiple Toasts appearing at the same time, pausing the timeout on hover, or adding a dismiss button, would not be a massive challenge with this setup.

Now imagine how hard these features can be if you are managing everything inside of a thunk. The moral of the story is: you don't need to use Redux for everything! There are cases where the more elegant solution resides on the component's side. It *is* OK to use local component state if that state is, in fact, local.

Using Redux, it is sometimes easy to fall into the habit of trying to solve every state-related problem with it. But one of the cool things about the React/Redux stack is that you *don't have to* do that! You *can* use whatever is the best tool for the job in that given situation. This is the upside of the high modularity the React/Redux world offers. Please use it!

Cleanup

We are almost ready to wrap up, but if you look at our current state tree, you will surely notice that it is looking a bit wonky. We have completely unrelated data living right next to each other inside one object. Entity data (`byId` and `ids`) sits right next to global ui state (`openNoteId`, `toast`) and network state (`requests`). We should clean this up.

Based on the categories we have identified above, our newly devised state should look like this:

```
{
  notes: {
    byId: {},
    ids: [],
  },
  ui: {
    openNoteId: null,
    toast: null;
  },
  requests: {},
}
```

Refactoring state shape is almost surprisingly simple. Changing the shape of the root reducer is almost as easy as writing an object literal:

```
// source/store/reducers.js
// ...

export default combineReducers({
  notes: combineReducers({
    byId,
    ids,
  }),
  ui: combineReducers({
    openNoteId,
    toast,
  }),
  requests,
});
```

And updating selectors is not much harder.

```
// source/store/selectors.js

export const getNotes = (state) =>
  state.notes.ids.map((id) => state.notes.byId[id]);

export const getNote = (state, id) =>
  state.notes.byId[id] || null;

export const getOpenNoteId = (state) =>
  state.ui.openNoteId;

export const getToast = (state) =>
  state.ui.toast;

// ...
```

Conclusion

And that's a wrap! Our app now allows users to feel in control, and thanks to Redux we now communicate network request states to make this happen. In the next chapter we will add the ability to our app to easily persist state across reloads.

7 Persisting State on Client

We now have an opportunity to demonstrate one of the benefits of using a Redux style architecture for managing state, which is the ability to easily persist state across reloads. Persisting state across reloads allows you to simplify your development workflow by reducing the need to recreate state to test features you are working on.

For example, let's imagine you are building a simple counter application that allows users to increment a count on your application by clicking a button. Whenever the count reaches 10 a modal should popup in the application to inform the user. However this logic isn't working and needs to be debugged. Traditionally we'd need to manually recreate the state to test the application, when we think we've fixed the issue we reload the application and recreate the state (by clicking the button 10 times) and then test the fix. We repeat this process until we've fixed the issue, thus reloading the page and therefore recreating the state after each change.

With Redux we can easily persist the state across sessions, allowing us to easily improve this workflow by providing the user with the option of saving the state of the application. Now when the user is attempting to fix this issue, they only need to create the required state once, and then save that state. Now each time you want to test a fix you simply reload the page and the state is restored, allowing you to skip the arduous task of recreating the state of the application and instead focus on fixing the issue. This is a big win for developer satisfaction.

This is great for development purposes, but should be used with caution in production applications. Usually this feature is restricted to the development environment.

How much to persist

When persisting data, you are faced with the choice of what to persist. Do you persist the entire state tree or do you persist only a few slices? There's no easy answer to this question, and it will depend on your specific use case. For development purposes it's often best to persist everything, but for application features you usually want more control so that you save only the relevant stuff and keep the state clean.

Automatic vs manual persistence

There are two approaches to persisting state: manual persistence and automatic persistence. Manual persistence requires the user to manually persist the state when they want it to be saved. This requires effort from the user and generally requires a little more effort to implement as well because you need to provide an interface to the user for persisting the data. Automatic persistence, on the other hand, requires no effort from the user, but also offers no control over when state is persisted.

Manual persistence

We're now going to implement our own manual persistence solution to allow the user to persist state when they choose. We're going to keep it simple, we won't be developing a UI and we won't be persisting specific state slices.

Persisting state

The first step is to persist the state. We could do this automatically every time the state is changed, or we can allow the user to manually save state through the UI or a public API. For the sake of simplicity we're going to use a public API that the user can interact with using the console, using something like `window.saveState()`.

Create a new file called `source/persist-client-state.js` and add the `saveState` function:

```
const saveState = (store) => {
    try {
        const state = store.getState();
        const jsonState = JSON.stringify(state);
        localStorage.setItem('state', jsonState);
        console.log('Application state saved to localStorage');
    } catch (e) {
        console.log('Unable to write to localStorage');
    }
};
```

This function accepts the Redux store as a parameter and saves the state from the store to local storage. There's nothing complicated happening here. Notice that we're wrapping the localStorage call in a try block because users sometimes disable localStorage which can cause exceptions to be thrown. We're going to call the `saveState` function through the window object, so we need to write the code to enable that:

```
export const initSaveState = (store) => {
    window.saveState = saveState.bind(null, store);
}
```

This function should only be called once, during the application initialization stage. We pass the store object to this function because it's a dependency of the `saveState` function. We declare a new `saveState` method on the `window` object and set it's value to a partially applied function that has it's first parameter bound to `store`. This means that calls to `saveState` don't have to pass the `store` object as it has already been supplied. Let's hook this up so we can save the application state. Open up `source/store/index.js` and add the following code:

```
import { createStore } from 'redux';
import rootReducer from './reducers';
import { initSaveState } from './../persist-client-state';

export default function configureStore() {
  const store = createStore(rootReducer);
  initSaveState(store);
  return store;
}
```

We're now in a position where we can save our application state with a single method call, and it only took us a couple of lines of code. Why not give this a try. Launch the application and add some notes. Now open the developer console and type the following:

```
window.saveState();
```

You should see a message printed in the console informing you that the application state was saved to `localStorage`. You can double check this by typing the following command to check `localStorage` yourself:

```
localStorage.get('state');
```

Restoring state

Now that we can save our state to local storage we need to add code to restore the state after reloads. Open up `persist-client-state.js` and add the `loadState` function:

```
export const loadState = () => {
    try {
        const jsonState = localStorage.getItem('state');
        if (!jsonState) {
            return undefined;
        }
```

```
            console.log('Application state restored from localStorage');
            return JSON.parse(jsonState);
        } catch (e) {
            console.log('Unable to load from localStorage');
        }
        return {};
    };
```

Now we just need to hook up the loadState function to the application initialization step. We've already enabled this by exposing the loadState method as part of this module's API, using the export keyword when declaring the loadState method. We can now use this method to load the persisted state and return it to the caller. Open up source/store/index.js and add the following code:

```
import { createStore } from 'redux';
import rootReducer from './reducers';
import { initSaveState, loadState } from './../persist-client-state';

export default function configureStore() {
  const store = createStore(rootReducer, loadState());
  initSaveState(store);
  return store;
}
```

That's all there is to it. We've now implemented client side state persistence, allowing you to save the current state of the application across page reloads. Let's give this a quick try.

Open the app and create some new notes. Now open the console and type window.saveState(). You should see a confirmation message confirming that the state has been saved. Now reload the page. You should see the application start using the same state you saved.

Automatic persistence

We can also automatically persist the application state, which results in less cognitive work for the user. This has a different set of use cases and is most beneficial for times when you don't need full control over when state is persisted.

Persisting state

Let's refactor our manual persistence solution to implement automated persistence. To begin, open up source/persist-client-state.js and export the saveState method. To do this, we simply just need to add the export keyword to the function declaration:

```
export const saveState = (store) => {
    try {
        const state = store.getState();
        const jsonState = JSON.stringify(state);
        localStorage.setItem('state', jsonState);
        console.log('Application state saved to localStorage');
    } catch (e) {
        console.log('Unable to write to localStorage');
    }
};
```

Now we need to import this function into the `configureStore` method. Open up `source/store/index.js` and update the import statement to include `saveState`:

```
import { initSaveState, saveState, loadState } from './../persist-client-
state';
```

Now we just need to subscribe to the store and call the `saveState` method. Add the following code just below the `initSaveState(store)` line.

```
store.subscribe(() => saveState(store));
```

This results in the following `configureStore` method:

```
export default function configureStore() {
  const store = createStore(rootReducer, loadState());
  initSaveState(store);
  store.subscribe(() => saveState(store));
  return store;
}
```

This one-liner simply subscribes to store updates using the subscribe method, anytime an update occurs it calls the `saveState` function passing the new state to it.

You can now load up the app and start modifying state (e.g. adding/editing notes). If you refresh the browser the current state of the application will be persisted automatically for you.

Refactoring

Really this is all you need to do, but this has left us with some obsolete code so we're not finished yet. We no longer need to pollute the window object to expose the `saveState` method, because it's now called called automatically for us after each state change. So let's remove this code. Open up `persist-client-state.js` and remove the following:

```
export const initSaveState = (store) => {
    window.saveState = saveState.bind(null, store);
}
```

We also need to remove the import and call from `source/store/index.js`. Open the file and remove the import:

```
import { saveState, loadState } from './../persist-client-state';
```

And finally remove the call:

```
export default function configureStore() {
  const store = createStore(rootReducer, loadState());
  store.subscribe(() => saveState(store));
  return store;
}
```

That's it! We've now refactored the existing code to remove the obsolete manual persistence logic.

Production Use

Unless you're using automatic state persistence as an application feature, you probably don't want this to be enabled in production builds. The easiest way to disable this feature is to remove module usage and allow tree shaking to remove the module from your build. This assumes tree shaking is supported by your bundler; tree shaking is a technique used by bundlers to remove unused code from your builds.

If your bundle tool doesn't support tree shaking, or you want a solution that is a little less manual, you can also use an environmental variable to determine when you are in production.

For example, to disable state persistence for the manual approach:

```
import { createStore } from 'redux';
import rootReducer from './reducers';
import { initSaveState, loadState } from './../persist-client-state';

export default function configureStore() {
  let defaultState = undefined;
  // Only hydrate using the persisted state in development
  if (__DEV__) {
    defaultState = loadState();
  }
  const store = createStore(rootReducer, defaultState);
  // Only allow the user to persist state in development
  if (__DEV__) {
    initSaveState(store);
  }
  return store;
}
```

Here, we're using the __DEV__ global to determine if we're in development mode. We're achieving this through the use of Webpack magic globals, using the `DefinePlugin` to set the __DEV__ global during development. This replaces the __DEV__ global with an actual value during build, resulting in output like this:

```
  if (true) {
    defaultState = loadState();
  }
```

Doing this isn't overly important with the manual solution we implemented because there is no visible UI. The only way users can persist state is by typing a command into the console. However, it's still worth doing, because you may forget and while testing your production build, you could be restoring persisted state without knowing. This could lead to some unexpected situations when testing your production builds.

Similar can also be used to disable state for the automated solution:

```
import { createStore } from 'redux';
import rootReducer from './reducers';
import { saveState, loadState } from './../persist-client-state';

export default function configureStore() {
let defaultState = undefined;
  // Only hydrate using the persisted state in development
  if (__DEV__) {
    defaultState = loadState();
  }
  const store = createStore(rootReducer, defaultState);
  // Only automatically persist state in development
  if (__DEV__) {
    store.subscribe(() => saveState(store));
  }
  return store;
}
```

These are simple and elegant solutions, but these solutions mean that the code isn't automatically removed from your build through tree shaking. Instead, you'll need to use an additional set of tools/plugins to enable this. For example, if you're using Webpack you can use a combination of the `strip-loader` plugin and the `NormalModuleReplacementPlugin` to achieve this behaviour.

Conclusion

We went through the process of manually implementing our own persistence solution, and while this is a great way to learn about the inner workings of this strategy, it's unnecessary because there are existing solutions available that can do this for us. One such solution is `redux-persist`, an open source library whose sole purpose is persisting state to different data stores.

In our example we stored everything in local storage, and while this provided a good solution for our needs, there are times when this won't suffice. `redux-persist` solves this problem by allowing you to change the data store that state is persisted to, for example allowing you to store state in a remote data store that enables users to persist state across devices, rather than being limited to the device where the state was created.

In the next chapter we cover middleware to intercept actions and send useful data to an analytics provider.

8 Analytics Middleware

A common requirement for any non trivial application is implementing analytics (such as Google Analytics) and tracking a user's behavior as they interact with the application. Redux's single state tree and strict unidirectional data flow makes this extremely easy to do. In previous chapters we covered the anatomy of middleware and discovered how easy it is to implement Redux middleware. In this chapter, we will leverage the power of middleware to intercept actions and send useful data to an analytics provider effectively enabling us to track user interactions at our discretion accross our sample application.

Why is this useful?

- This is a common real world requirement for most applications.
- It becomes super easy to update and maintain analytics as our app scales when using Redux middleware to manage our analytics tracking side effects.
- Analytics is colocated with user interactions. Because we will attach analytics data to our actions, it is easier to augment our tracking with relevant information about a user action.
- Analytics are centralized, making it extremely easy to determine what is being tracked and what is not, helping with maintainability.
- You can reuse the concepts outlined here for other third-party APIs and side effects.
- Your favorite business owners and analytics guys will love you!

Middleware Specification

- Allow centralized configuration of analytics tracking the API.
- Only intercept actions that declare analytics meta data.
- Allow multiple middleware instances that accept custom tracking functions.
- Enforce healthy sanity checks on our actions to prevent accidents.

Creating the middleware

We can reuse our `loggerMiddleware` as a starting point for our new analyticsMiddleware:

```
const loggerMiddleware = ({ dispatch, getState }) => next => action => {
  const prevState = getState();
  const result = next(action);
  const nextState = getState();

  console.group('Dispatch', new Date());
  console.log('Previous state', prevState);
  console.log('Action', action);
  console.log('Next state', nextState);
  console.groupEnd('logger');

  return result;
}
```

Gut our loggerMiddleware to the bare essentials and create a placeholder track handler for our analytics tracking API:

```
const track = (action, state) => {
    // analytics tracking api logic goes here
    console.log(`Tracking [${type}]:`, payload);
}

const analyticsMiddleware = ({ getState }) => next => action => {

  const result = next(action);

  track(action, getState());

  return result;
}
```

Add our track handler and limit our middleware to only intercept actions with declarative analytics meta data:

```
const analyticsMiddleware = ({ getState }) => next => action => {

  const result = next(action);

// if there is no declared analytics meta, let's not track
  if (!action.meta || !action.meta.analytics) {
      return result; // return to middleware chain
  }

  track(action, getState());
  return result;
}
```

Refactor to only pass relevant meta data to our track function:

```
const analyticsMiddleware = ({ getState }) => next => action => {

  const result = next(action);

// if there is no meta, return to middleware chain
  if (!action.meta || !action.meta.analytics) {
      return result;
  }

  const { type, payload } = action.meta.analytics;

  track({ type, payload }, getState());

  return result;
}
```

Here is configuring in our store:

```
import { applyMiddleware, createStore } from 'redux';
import reducer from './path/to/our/reducer';
import analyticsMiddleware from './path/to/our/analyticsMiddleware';
const store = createStore(reducer, applyMiddleware(analyticsMiddleware));
```

Augment actions

We augment our actions with declarative meta data to enable our `analyticsMiddleware` to track. This allows us to decide which actions should and should not be handled by our middleware and also pass along any relevant data to our middleware and tracking API.

```
// let's suppose we can attach common `tags` to our notes and we want to
measure how many times a specific userType is opening notes with a specific
set of `tags`
export const openNote = (id, userType, tags) => ({
  type: 'app/openNote',
  payload: { id },
  meta: {
      analytics: {
          type: 'app/openNote',
          payload: {
           id,
           userType,
              tags
            }
        }
    },
});
```

Further refactoring

Let’s change our middleware signature to allow custom track functions to be passed to middleware, making it more reusable. We will also add some nice checks to ensure the shape of our actions, follow a standard format to enforce consistency, and prevent issues accross many actions:

```
import { isFSA } from 'flux-standard-action';
// let's enforce our middleware to only respect flux standard actions, this
will help prevent mistakes as we scale

const analyticsMiddleware = (handler) => ({ getState }) => next => action =>
{
// allow for custom handler to be passed, we can then reuse middle ware if
we need to

  const result = next(action);

  if (!action.meta || !action.meta.analytics) {
  //if there is no meta, return to middleware chain
      return result;
  }

  if(!(isFSA(action) && isFSA(action.meta.analytics))){
      console.error('Actions that are not Flux Standard Actions will not be
processed by analyticsMiddleware');
      return result;
  }

  const { type, payload } = action.meta.analytics;

  handler({ type, payload }, getState());

  return result;
}


import { applyMiddleware, createStore } from 'redux';
import reducer from './path/to/our/reducer';
import analyticsMiddleware from './path/to/our/analyticsMiddleware';

const track = ({ type, payload }, state) => {
    // analytics tracking api logic goes here
    console.log(`Tracking [${type}]:`, payload);
};

const store = createStore(reducer, applyMiddleware(analyticsMiddle-
ware(track)));
```

Accessing our entire state within our track function

Our second argument is the entire state tree, allowing us to capture shared analytics data or perform conditional logic before calling our analytics API:

```
const track = ({ type, payload }, state) => {
    // analytics tracking api logic goes here
    console.log(`Tracking [${type}]:`, payload);
    // do some conditional logic with tracking api based on state or pass
shared data, example:  state.userType
}
```

Final implementation

Although the example below is implemented using the Adobe Analytics custom tracking API, we can reuse this middleware with any analytics provider:

```
import { applyMiddleware, createStore } from 'redux';
import reducer from './path/to/our/reducer';
import analyticsMiddleware from './path/to/our/analyticsMiddleware';

// _satellite is Adobe Analytics global analytics object

const analytics = analyticsMiddleware(({ type, payload }, state) => {
    try {
        window._satellite.setVar('payload', Object.assign({}, payload,
state.userType));
        window._satellite.track(type);
    } catch (err) {
        console.error(err);
    }
});

const store = createStore(reducer, applyMiddleware(analytics));

// ... inside of Adobe Analytics, we can configure rules to accept
these .track(type) calls and assign the `payload` sent with .setVar()
```

Conclusion

Here are some useful implementations of this pattern:

- **https://www.npmjs.com/package/redux-reporter**
- **https://www.npmjs.com/package/redux-analytics**
- **https://www.npmjs.com/package/redux-keen**
- **https://www.npmjs.com/package/redux-segment**
- **https://www.npmjs.com/package/redux-optimizely**
- **https://www.npmjs.com/package/redux-adobe-dtm**

Our sample app now allows us to track user interactions at our discretion. In the next chapter we will focus on Redux best practices.

Best Practices 9

The Redux ecosystem as a whole is flourishing and extensive. Redux's author, Dan Abramov, has put together some great documentation and the community has done an excellent job growing the Redux ecosystem. However, a lot of knowledge is still spread across many examples, tutorials, Github issues, and engineers. In this section we will cover a condensed version of useful lessons, FAQs, and tips learned from the trenches, and guide you around common questions and pitfalls when scaling a non-trivial real world Redux application. We will tackle the following:

- Actions
- Reducers
- Selectors
- Middleware, Store Enhancers
- Project Structure
- UI, Rendering
- Developer Tooling and Debugging
- Testing

Actions

In real world applications, simple synchronous actions are not enough to handle everything your app will do. You'll likely need to handle async behavior, deal with promises, trigger multiple actions, or dive into more complex recipes. Although there's no specific rule for how you should structure your actions and action creators, here are some general insights.

Disambiguation

- A Redux `action` is a plain old JavaScript object (POJO). It consists of a `type` field and optional data (often stored in a `payload` field) that describes a change for your application.

- An `action creator` is a function that creates an action. They do not dispatch to the store (unlike Flux), yet they return action objects and may contain additional logic to prepare an action object.
- Actions are the only way to get data into the store. Pass the result of an action creator to the `dispatch()` function to trigger a state change.
- Redux's `bindActionCreators` turns an object whose values are action creators, into an object with the same keys, but with every action creator wrapped into a dispatch call so they may be directly invoked.
- `Redux thunk` is an extremely useful middleware that uses thunks to solve common problems you'll likely encounter. It allows you to write action creators that return a function instead of an action. You can use thunks to delay the dispatch of an action, or to dispatch only if a certain condition is met. They are also used when creating Async Action Creators.

ACTIONS SHOULD BE SERIALIZABLE

Serializable actions are at the heart of Redux's defining features and enable time travel and replaying actions. It is okay to use Promises or other non-serializable values in an action as long as it is intended for use by middleware. Actions need to be serializable by the time they actually reach the store and are passed to the reducers.

DEFINING AND ORGANIZING ACTION TYPES

Action types should be defined as self-descriptive string constants helping to reduce repetitiveness, reproduce issues, and facilitate debugging.

Defining types in a constants module can help encourage you to think early on about how you want to organize your app. This can act as a manifest for all of the things that can occur in your app. This can be extremely useful when discussing the app with other devs (use the action list as a reference), or with other teams, such as discussing actions with your UX or analytics team.

Organizing action types by `export`:

```
// constants/ActionTypes.js
export const ADD_NOTE = 'ADD_NOTE';
export const DELETE_NOTE = 'DELETE_NOTE';
export const FAVORITE_NOTE = 'FAVORITE_NOTE';
```

Organizing action types with `keyMirror`:

```
// constants/ActionTypes.js

import keyMirror from 'keymirror';

const ActionTypes = keyMirror({
    NOTES_REQUEST: null, // yes! less typing
    NOTES_SUCCESS: null,
    NOTES_FAILURE: null

});

export default ActionTypes;
```

The advantage to organizing your action type constants with `keyMirror` is to reduce the likelihood of typos. The `keyMirror` module is also used in many traditional Flux examples (by Facebook), and perhaps may be familiar to you. If you are migrating a traditional Flux app, you do not have to change the way your action types are organized.

The downside is that it becomes a little more difficult to import your constants individually using ES6, since you'll have to import the entire object that was returned by `keyMirror`. Another downside is that it becomes more difficult for static analysis and IDEs to track down where constants are defined.

To organize action types along side your action creators:

```
// actions/ApiActions.js

// not going to forget to declare these constants in another file
export const NOTES_REQUEST = 'NOTES_REQUEST';
export const NOTES_SUCCESS = 'NOTES_SUCCESS';
export const NOTES_FAILURE = 'NOTES_FAILURE';

export function fetchNotes() {

    return (dispatch) => {

        dispatch({ type: types.NOTES_REQUEST });

        return fetch(url, options)
            .then(parseResponse)
            .then(checkStatus)
            .then(normalizeJSON);
            .then((json) => dispatch({ type: types.NOTES_SUCCESS, pay-
load: json }))
            .catch((error) => dispatch({ type: types.NOTES_FAILURE, pay-
load: error, error: true }));

    };

}
```

The advantage to organizing your action constants next to your action creators is to keep your code condensed with minimal duplication. This becomes useful if your app is fairly small, with only a small amount of action types. It also becomes easier to reference these constants when editing your action creators. The downside depends on the size of your application. If you have a lot of actions or complex action creators, such as middle-ware, it may be better to split your constants into a separate folder and/or files for easier reuse and smaller action creator file sizes. Of course, it depends on your app and your preferred file/folder structure.

Another downside to this organization is that your reducers need to access action constants defined next to your action creators. If you want to prevent duplication, they need to be exported, and your reducers would then import them directly from your action creator modules. At this point, perhaps it is easier to keep them in a seperate file.

In order to import your constants:

```
    // actions/ApiActions.js
    import ActionTypes from '../constants/ActionTypes';

    const { NOTES_REQUEST, NOTES_SUCCESS, NOTES_FAILURE } = ActionTypes;

    //  Will I end up duplicating anyways?

    function createFetchActionCreator(options, types){
        //deconstruct
        const [NOTES_REQUEST, NOTES_SUCCESS, NOTES_FAILURE] = types;
        //...
    }

    const fetchNotes = createFetchActionCreator(options, [NOTES_REQUEST,
NOTES_SUCCESS, NOTES_FAILURE]);
    // ...
    dispatch(fetchNotes());

    // actions/ApiActions.js
    import ActionTypes from '../constants/ActionTypes';
    // is the entire object needed?
    // ...
    dispatch({ type: ActionTypes.NOTES_REQUEST });

    // Does repeating the object become ugly?

    // actions/ApiActions.js
    import { NOTES_REQUEST, NOTES_SUCCESS, NOTES_FAILURE } from '../
constants/ActionTypes';

    // ...
    dispatch({ type: NOTES_REQUEST });

    // this is specific
```

Keep in mind that the way you organize your constants can make it easier to import. Consider the fact that you may have to import in several places, such as within your custom middleware and your reducers. Depending on how you organize these, you may only want to import one or two action types, as opposed to an entire object, and benefit when optimizing by being specific on what action types are actually used within your module.

ACTION NAMING CONVENTIONS

Naming actions is extremely important, because your actions describe what your app can do and what your action does. As your app grows, the easier it is to understand what an action does, and the easier it will be to quickly traverse your application. When naming actions (the action type constant and action creator) it can be helpful to derive the name of an action directly from a sentence that describes what you want your app to do (creating statements of functionality).

```
//When the page loads, we want our app to Fetch notes
//becomes:  FETCH_NOTES, dispatched on page load

//When a user clicks the checkbox it will Filter the notes list
//becomes:  FILTER_NOTES, dispatched when user checks the checkbox

//The radio button will Toggle the visibility of favorited notes
//becomes:  TOGGLE_NOTES_VISIBILITY, dispatched when the radio button is tog-
gled
```

Avoid going beyond three segments in your action name. This may be a first sign that something else in your app needs to be fixed or addressed. Make sure that it is clear on what your action does. Being too vague can be painful for other developers jumping into the project. If you are doing it right, you should not have to comment or explain your actions.

Good:

```
    export const ADD_NOTE = 'ADD_NOTE';
```

Bad:

```
    export const NOTE = 'NOTE'; // fine, we have notes, but am I adding,
deleting, or favoriting?
```

Bad:

```
    export const ADD = 'ADD';  // add what?
```

Be careful not to use terms that may mean something else within your app.

Good:

```
    export const RESET_ERRORS = 'RESET_ERRORS';
```

Bad:

```
    export const RESET_STATE = 'RESET_STATE';  // reset the entire state
tree?  or just the errors?
```

Your actions should be specific and typically only describe doing one thing at a time. This makes it easier to understand, and follows the single purpose rule, making it easier for add/deleting and refactoring. Keep in mind that you may have actions that result in multiple things occuring. Multiple reducers can listen to the same action and the intention of the action may actually impact multiple parts of your state; however, we should still aim to name our actions in a singular manner to avoid confusion and to maintain consistent verbage.

Good:

```
    export const DELETE_NOTE = 'DELETE_NOTE';
    export const SHOW_FORM = 'SHOW_FORM';
```

Bad:

```
    export const DELETE_NOTE_AND_SHOW_FORM = 'DELETE_NOTE_AND_SHOW_FORM';
```

Good:

```
    export const LOAD_PROJECT = 'LOAD_PROJECT';
    // handled my multiple reducers and many things occuring when dispatched
```

Bad:

```
    export const LOAD_PROJECT_AND_LOAD_ENTITIES = 'LOAD_PROJECT_AND_LOAD_EN-
TITIES';
 // do not describe implementation details or what your reducers are expect-
ed to do in the action name.  Describe the intention of the action.  If it
is not clear, consider introducing another action.
```

CREATING ACTIONS, TYPE WITH FLAGS VS MULTIPLE ACTION TYPES

It is important to decide on a convention early on with your team.

You may be inclined to reuse an action or a type for multiple things and attach a dedicated status field in your actions (Bad):

```
{ type: 'FETCH_POSTS' } //bad, not sure if this is the start or end of async
flow, need to type check
{ type: 'FETCH_POSTS', status: 'error', error: 'Oops' } //bad, now my reduc-
er needs to figure out more, and this isn't a standard pattern
```

```
{ type: 'FETCH_POSTS', status: 'success', response: { ... } } // does not
follow Flux Standard Action, may need to normalize in multiple places
```

Although this does work for trivial applications, it can become confusing as you scale, especially during debugging. Imagine what your Redux devtools list/logs will look like with these action types. You now have to expand and examine every single action every time this action is dispatched to determine if it is the start, error, or success. It may cause more conditions and type checking within your reducers, making it likely that you will need to duplicate, especially if logic in your actions are used in multiple reducers. It also becomes more difficult to write middleware or reuse other's middleware because it does not follow a standard pattern.

You can and should define separate types for distinct actions, even if it feels verbose. Multiple types leave less room for a mistake, make it easier to debug, and reduce ambiguity (Good):

```
{ type: 'POSTS_REQUEST' } // nice, I can use this to toggle a loader on
{ type: 'POSTS_FAILURE', error: true, payload: new Error('oops...') } // i
can check the error field within my error handling reducer very easily, I al-
so have access to the error
{ type: 'POSTS_SUCCESS', payload: { ... } } // if i only care about handling
data, I know exactly what action to listen to, in my middleware I can also
spot types postfixed with _SUCCESS
```

As your app grows it is very obvious what each type does. It is also very easy to reuse the action across multiple reducers and easily understand them as context may be different in different reducers. For instance, POSTS_SUCCESS in your posts.js reducer may be used to update your list of posts in the post list component. It may also be used in the navigation reducer and navigation component to update the count pill on the navigation. Finally, remember that others will likely be working on the project, so defining your naming conventions early on can facilitate when devs copy and paste, or need to quickly ramp up on a new section of code.

Examining the types:

```
{ type: 'POSTS_REQUEST' }
```

An action informs reducers that the request began. Reducers can use this to toggle on a loader (`isFetching` flag in your state).

```
{ type: 'POSTS_FAILURE', error: true, payload: new Error('oops...') }
```

An action informing reducers that the request failed. Reducers can use this to reset `isFetching` state or display error messages.

```
{ type: 'POSTS_SUCCESS', payload: { ... } }
```

An action informs the reducers that the request finished successfully.

USE FLUX STANDARD ACTIONS (FSA'S)

Although it is not required, as actions only require a type, it is recommended to use the Flux Standard Action format for your actions (also called FSA's). FSA's define simple guidelines for a common format. See the official documentation: **https://github.com/acdlite/flux-standard-action**

It is much easier to debug and work with actions if we use a common format. Using a common format can facilitate sharing code or extracting logic into middlewares. We can also use the flux-standard-action module to ensure our actions are compliant.

In a real world app we can create FSA compliant actions by convention, and use FSA's `isFSA` helper function in our tests for verification. FSA's are extremely useful when creating middleware. Because we are using a standard, we can create middleware that depends on particular action properties and write reliable and efficient code to handle these properties. A common example is intercepting meta data attached to our actions within Analytics Middleware mentioned in previous chapters.

SHARING STATE BETWEEN TWO REDUCERS

If you need to share data between two reducers, the fastest (typical way) is to use Redux thunk and access state through `getState()`. An action creator can retrieve additional data and put it in an action so that each reducer has what it needs to update its own slice of state. Notice that the inner function receives the store method dispatch and `getState` as its parameters.

```
function addNote(){
  return (dispatch, getState) => {
  // i can get needed state for my action to make sense without passing more
props

    // i have access to the entire state, if i wanted, i could reuse a selec-
tor here
    const { name } = getState();

      dispatch({
          type: ADD_NOTE,
          payload: {
              note: 'my note',
              author: name
          }
      });
  };
}
```

```
store.dispatch(addNote());
```

General insights:

- There are additional ways to share data between two reducers. Another common method is to use top level reducers that have access to other reducers. Depending on your needs, this may be a better choice.
- Perhaps it is better to provide your component with the necessary props and pass to your action creator if they need them anyways.
- Depending on your preference, you may not want your action creators to know too much. In this case, you may have to opt for an alternate technique.
- Always be concious of timing issues when coordinating state between multiple reducers.
- See the official Redux FAQ for more indepth information on sharing state between reducers: **http://redux.js.org/docs/FAQ.html#reducers-share-state**.

PERFORMING CONDITIONAL DISPATCH LOGIC WITH THUNKS

Generally, reducers contain the business logic for determining the next state. However, reducers only execute after actions are dispatched. Thunks allow you to read the current state of the store and subsequently perform conditional dispatches before your actions reach your reducers. It is best to use this pattern if you are not able to perform the logic within your reducers, or you would like to extract out complex logic from your view.

```
function incrementIfOdd() {
  return (dispatch, getState) => {
    const { counter } = getState();
    if (counter % 2 === 0) {
      return;
    }
    dispatch(increment());
  };
}

store.dispatch(incrementIfOdd());
```

General insights:

- It can be better to handle this type of logic in your reducer, depending on other state and where you want to put your logic. Perhaps you have multiple reducers that need to listen to an intended increment action and handle accordingly.
- Leveraging action creators (thunks) to solve complex logic is often an easy choice and very helpful when coordinating issues where timing is critical.

- There is not a hard and fast rule that says all your business logic needs to be contained only in your reducers, or only in your actions. A mixture of both, depending on your preference or needs, is acceptable.

TIPS

- You can use action creators to transform API data. Using action creators to transform UI data to what makes sense for your APIs is common. This works for both requests and responses. Because your reducers only receive actions, they are unaware of how the action was created. You will likely have to deal with legacy APIs, APIs out of your control, poorly designed APIs, or APIs not specific to your application. In these cases the responsability can be given to action creators to transform responses and requests of an API call to make sense for your reducers.
- Action Creators can be asynchronous (think redux-thunk middleware).
- Since your application will require asynchronous updates, business logic or conditional logic may end up in your action creators, however, remember that actions need to be serializable by the time they actually reach the store and are passed to the reducers.
- Although it is okay for your reducers to contain logic that transforms your data, you will want to avoid duplication depending on the action and data. If multiple reducers need to receive a payload that is ready or you find your reducers caring about other slices of state than it's own, put the logic to prepare the data in your action creators, not your reducers.
- Action creators can share selectors, because they have access to the complete state with thunks. Reducers often can't share selectors, because they only have access to their own slice of state depending on how they are organized.
- Using thunks, action creators can dispatch multiple actions, making complicated state updates easier and encourages code reuse.
- Without middleware, Redux only supports synchronous data flow. Async middleware (redux-thunk, redux-promise) wraps `dispatch()` and allows you to dispatch something other than actions, for example, functions or Promises. You can use Redux thunk and Redux Promise together; they are not mutually exclusive.
- Putting constants in a separate file allows you to check your import statements against typos to prevent accidentally using the wrong strings.
- If you find yourself duplicating code accross multiple action creators, consider condensing the logic into middleware.

Reducers

We use reducers to shape and organize our state. When you embark on more complex applications, designing and organizing your reducers is not always trivial. In previous chapters we covered the art of designing your state and how to dissect UI state and domain specific state slices. Here are some tips that can help facilitate creating your reducers and keeping your state design clean.

General insights:

- Reducers take the previous state and an action, returning the next state. This is an important understanding when deciding how to handle data and data mutations within your reducers.
- Reducers are synchronous pure functions with no side effects, this makes them extremely predictable and easily testable.
- Typically, reducers group the various slices of application state and become the top level keys within your state tree.
- There is not a hard and fast rule on where business logic should reside (inside action creators or inside reducers), but the key is to find a balance that makes sense for your application.
- Reducers do not need to exclusively return objects. They can return arrays, numbers, strings, or whatever object shape that makes sense for a particular reducer (slice of state).

Reducer initial state

Creating a reusable `initialState` const can be useful for reuse or returning back to the initial state of the reducer. A well-defined `initialState` can also act as a psuedo schema for your reducer. In other words, you create your `intialState` const and all possible properties with default values, allowing it to act as a point of reference for what your resulting state will look like. This is particularly useful for other devs jumping into the project, because they can glance at the initial state before diving into complex logic. Of course, sometimes this is not always possible, but it is good to do if possible. Exporting the `initialState` const can also be useful to import into your tests.

```
export const initialState = {  // I know exactly what my reducer will
change, I also export for use in my tests
    isFetching: false,
    notes: [],
    message: 'Your notes list is empty, click to fetch them.',
    numberOfVisibleNotes: 0,
    author: null  //updated when user inputs into field
};
```

ELIMINATING SWITCH STATEMENTS

Common reducer examples use the switch statement to handle actions. This is not a requirement, as it is okay to use if statements or use other conditional logic that matches by something other than type (such as logic based on payload data).

```
const initialState=[];
// ...
```

```
function meta(state = initialState, {
    type,
    payload,
    meta
}) {

    if (meta.something) {
        return state.concat(payload);
    }
    return state;
}
```

There are packages such as `redux-actions` that contain helper methods to facilitate in eliminating large switch statements. In general, if you find your reducers with extremely large switch statements, consider extracting logic into helper functions or dividing your reducer into multiple smaller reducers.

TIPS

- Reducers are just functions, so you can use functional composition and higher-order functions to compose your reducers.
- `redux-actions` package contains several helpers to simplify creating actions and reducers. This may be a viable addition depending on your needs. **https://github.com/acdlite/redux-actions#handleactionsreducermap-defaultstate**.
- It is perfectly acceptable to detach a reducer from a store. Reducer functions could be used in other places besides a Redux store. For example, you could use one to calculate updated local state for a React component.
- Common practice is to divide state into multiple slices or domains by key, with a seperate reducer function to manage each slice. Consider naming reducers by function, not the domain they serve.
- Data from the server (an API) is not necessarily your final model. Your final state tree will be composed of multiple data inputs. It is okay if your reducers do not align perfectly with the structure of your API(s).
- Splitting your store into multiple smaller reducers allows for separation of concerns and can also eliminate large switch statements. It can also facilitate testing and reduce cognitive overload when traversing your application.
- Multiple reducers can listen to the same action.
- Using multiple smaller reducers allows us to use libraries (such as Immutable.js) on a per reducer basis; if you do not need a particular library for your entire state tree, you don’t have to use it for every reducer. This can help with refactoring, transitioning to new libraries, or porting over functionality from other applications.
- Naming your reducer files the same as your reducer key can help facilitate debugging and quickly finding troublesome logic.
- Reducers should not dispatch actions.

- If you have reducers that need to depend on other state, you can pass it explicitly as the third argument to your reducers.
- You can use `combineReducers()` multiple times, since it returns a reducer with (state, action) => state signature, just like a standard reducer.
- It is okay to use additional functions in the reducer structure to help break things into smaller pieces.
- If you have deeply nested data structures, consider using libraries such as `normalizr` or `redux-orm`, see the official Redux FAQ for more information: **http://redux.js.org/docs/FAQ.html#organizing-state-nested-data**

Selectors

A selector is a function that selects part of the state tree. Selectors are also commonly used to return data that is derived from the state.

Tips

- Redux state should always be your source of truth.
- Generally, it is not a good idea to put derived data (such as filtered state) in the Redux store. Using selectors to compute derived data allows for reuse across components and also allows Redux to store the minimal possible state
- Selectors simplify testing and facilitate extracting complex logic out of your components.
- Reselect is a recommended selector library to write performant selectors. It is extremely powerful when projects have performance problems because of many derived computations causing multiple re-renders. Reselect creates memoized functions that will return the last value without recalculating if called with the same arguments multiple times in a row.
- Colocating selectors with reducers helps organize related logic. A common practice is to make your default module export your reducer, and a prefix named selector exports with `get`.
- Official `Reselect` docs: **https://github.com/reactjs/reselect**.

Middleware, store enhancers

Middleware provides a third-party extension point between dispatching an action and the moment the action reaches the reducer.

A store enhancer is a higher-order function that composes a store creator to return a new, enhanced store creator. This is similar to middleware in that it allows you to alter the store interface in a composable way.

Tips

- Use middleware to handle common side effects like analytics tracking.
- Redux thunk has a helper `withExtraArgument` so you can inject extra arguments allowing you to mock your async API making writing unit tests easier. This can be useful to inject common functionality.
- You can use middleware to validate state across reducers, creating a single point that can analyze your entire state tree.
- More on store enhancers: **https://github.com/reactjs/redux/blob/master/docs/Glossary.md#store-enhancer**.

Project Structure

While Redux itself has nothing to say about file structures (as it should depend on your specific needs), there are a number of common patterns for organizing the file structure of a Redux project. Each pattern has pros and cons. Let's explore these pros and cons.

Organizing by Concept Type

The most common way is to organize your project by concept. This is simply arranging the application files by function. Each concept lives under a common parent.

```
├── src                        # source
│   └── styles                 # css/sass/less
│   └── images                 # .png,.svg,.gif etc.
│   └── js                     # react/redux source
│     ├── actions              # redux actions
│     ├── components           # react components/containers
│     ├── constants            # constants, action types
│     ├── reducers             # reducers
│     ├── selectors            # selectors, using reselect
│     ├── store                # store, w/devtools & prod config
│     ├── utils                # utilities/helpers
│     └── index.js             # app entry
│   └── index.html             # app shell
├── test                       # tests
│   └── *.js                   # specs
│   └── setup.js               # test config
├── webpack                    # webpack
├── .babelrc                   # babel config
├── .eslintrc                  # eslint config
├── .gitignore                 # git config
├── LICENSE                    # license info
├── package.json               # npm
└── README.md                  # installation, usage
```

PROS

- Easy organization
- Follows many tutorials and examples
- Easy to get started on trivial applications
- Does not couple types
- Those new to a project may initially find it easier to traverse
- No duplication in file names

CONS

- Repetition of module groupings/names
- Splitting up application for optimization is more difficult
- No group of related features
- Naming features can be more difficult
- Editing a feature means traversing multiple files/folders

Organizing by Feature or Domain

Another common way to organize your project is by arranging your files into feature folders, grouping everything related to a specific feature.

```
├── src                             # source
│   └── notes                       # feature
│     ├── Notes.js                  # component/container
│     ├── Notes.test.js             # tests
│     ├── NotesActions.js           # actions
│     ├── NotesActions.test.js      # tests
│     ├── notesReducer.js           # reducer/selectors
│     └── notesReducer.test.js      # tests
│   └── comments                    # feature
│     ├── Comments.js               # component/container
│     ├── Comments.test.js          # tests
│     ├── CommentsActions.js        # actions
│     ├── CommentsActions.test.js   # tests
│     ├── commentsReducer.js        # reducer/selectors
│     └── commentsReducer.test.js   # tests
│   └── index.html                  # app shell
├── webpack                         # webpack
├── .babelrc                        # babel config
├── .eslintrc                       # eslint config
├── .gitignore                      # git config
├── LICENSE                         # license info
├── package.json                    # npm
└── README.md                       # installation, usage
```

PROS

- Feature naming is clear
- Architecture/structure independent
- Trivial to split code (lazy load features based on routes or conditions)
- Easier to pluck features out of application for reuse
- Allows for feature encapsulation and ability to expose API using a root index.js to export your reducers, action creators, and selectors

CONS

- Groupings must be repeated
- May be overkill depending on project size
- Still requires central store configuration
- May encourage mapping your data to a single feature
- Not everything may belong to a specific feature
- Sharing functionality accross features may be ambiguous and awkward

Alternatives

Depending on your project's complexity, you may want to choose to group your actions and reducers in one file. It is a slight variation that may help readability, particularly in smaller projects. The community already took note of such a pattern and Erik Rasmussen summarised the approach under the concept of Ducks **https://github.com/erikras/ducks-modular-redux**. According to him, it makes more sense for these pieces to be bundled together in an isolated, self-contained module which can be easily packaged into a library.

General Insights

- Most of the time components are not used outside of containers.
- ES6 allows to export multiple items, which allows you to merge two things into one file and use export default as the container and regular export the component.
- Configuring test runners is trivial for both patterns.
- Reducers rarely correspond to specific features/pages in a clear way.
- Choose the pattern that makes sense for your needs, there is no right or wrong way to organize your application, consistency is key.
- Your project structure is usually different from state shape, do not feel compelled to shape your state tree based on your feature/file organization.
- Focus on structuring your reducers, actions, and action creators well. If needed, they can always be moved to alternate project structures.

UI, Rendering Tips

- react-redux's `Provider` HOC (higher order component) wraps a root component and makes it possible to use `connect()`
- When using react-redux, all commonly discussed react optimizations still apply
- A good pattern to adopt is the 'Presentational and Container' component pattern. **https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0**.
- As your app grows/matures the components that are connected to Redux are likely to change. Embrace the idea that you may need to refactor later on and eliminate container components, or connect child components.
- Prefer smaller stateless functional components when possible.
- There is not a hard and fast rule that says every piece of state should belong in the Redux store, it is okay for components to manage thier own state (using reacts state management).
- `recompose` is a handy set of React tools to optimize your components **https://github.com/acdlite/recompose**.
- `onlyUpdateForKeys` from `recompose` can be used to make sure only updates occur if specific prop keys have changed
- If you need to apply several HOCs on a component you can use Redux's `compose()` as follows:

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

class MyComponent extends Component {
    // ...
}

const enhance = compose(
    onlyUpdateForKeys(['propA', 'propB']),
    connect(mapStateToProps, mapDispatchToProps)
    // anything else
)

export default enhance(MyComponent)
```

Developer tools

The two most useful tools for developers and debugging Redux applications are custom logging middleware and the `redux-devtools-extension` package for Chrome, Firefox, and Electron. These tools will cover your needs 99% of the time. The custom logging middleware allows you to tweak your logging as you go and has a simple, small footprint.

The `redux-devtools-extension` allows for minimal configuration to enhance your Redux store and leverages the power of several existing Redux DevTools extensions. This makes it easy to implement and easier to maintain (it is not directly coupled to your application code).

Custom Logging middleware:

```
// logger.js
export default (store) => (next) => (action) => {
    console.groupCollapsed(action.type);
    console.info('action:', action);

    const result = next(action);

    console.debug('state:', store.getState());
    console.groupEnd(action.type);

    return result;
};
```

Redux DevTools Extension: **https://github.com/zalmoxisus/redux-devtools-extension**

```
// configureStore.js
import {createStore, applyMiddleware, compose} from 'redux';
import rootReducer from '../reducers';
import thunk from 'redux-thunk';
import logger from '../middleware/logger';
const IS_BROWSER = typeof window !== 'undefined';
const IS_DEV = process.env.NODE_ENV !== 'production';
const middleware = [thunk];

if(IS_DEV){
    middleware.push(logger);
}

export default function configureStore(initialState = {}) {

    return createStore(rootReducer, initialState, compose(
        applyMiddleware(...middleware),
        IS_DEV && IS_BROWSER && window.devToolsExtension ? window.devToolsEx-
tension() : (f) => f
    ));

}
```

For additional developer tools & resources, checkout out: **https://github.com/markerikson/redux-ecosystem-links/blob/master/devtools.md**

Testing Tips

- For async action creators, it is best to mock the Redux store, so redux-mock-store is extremely useful.
- When testing action creators, we want to test whether the correct action creator was executed and also whether the correct action is returned.
- When using thunks with promises, you should return your promise for testing. If you are using redux-thunk and fetch, it is recommended to return your fetch promise from your action creator.
- If you have adopted a standard action shape (perhaps Flux Standard Action), you may test that your action creators follow the standard. You may also want to enforce FSA's with middleware.
- Testing a reducer should always be straight forward, reducers simply respond to incoming actions and turns the previous state to a new one. If you find testing reducers difficult, it is likely you are doing something wrong within your reducers.
- In general, if we avoid using large reducers and compose our final store with multiple smaller reducers, our tests become easier to organize in addition to the benefits outlined in the Reducer best practices section.
- Generally having 1 test file for every 1 reducer file makes it easier to navigate your tests as your app grows. It also allows you to reorganize your project if needed. For example, perhaps you decide to refactor your app and move your reducers into modules, you can simply move the corresponding test.
- You can export an `initialState` value from your reducer modules to reuse in your tests.
- Use action creators within your reducer tests to avoid duplication of action objects.
- When testing `mapStateToProps()`, extract behavior from `mapStateToProps()` and move into selectors, since selectors are easier to test.
- Selectors should be pure functions.
- `mapDispatchToProps()` is usually not worth testing.
- For more testing resources, checkout **https://github.com/markerikson/react-redux-links/blob/master/react-redux-testing.md**.

Conclusion

The tips in this chapter will help you as you implement your own Redux applications. In the next and final chapter, we explore the topic of going offline.

Going offline 10

In order to understand why we should care about the offline accessibility of our apps, we have to travel back in time a bit. Some good memories will hopefully come together with our trip.

Somewhere between 2002 and 2005 people started building what we would today know as Single Page Applications (src: **https://en.wikipedia.org/wiki/Single-page_application**). At the time, JavaScript was a bit of a mess and was starting to consolidate into ECMA 4 (**https://en.wikipedia.org/wiki/JavaScript#Standardization**). IE had the largest market share (ref: **https://en.wikipedia.org/wiki/Usage_share_of_web_browsers**) and Firefox had just seen the light in 2004 (src: **https://en.wikipedia.org/wiki/History_of_Firefox#Version_1.0**).

UX wasn't a thing back then. However, some brave web developers started to tinker with a mad idea: what if the browser would actually run client logic and we wouldn't have to go through hops in server hard-reloads? And so, people rediscovered XMLHttpRequest and AJAX was born. It was indeed a revelation!

It wasn't until 2008 that Google Chrome came into existence. Can you believe that? However, 2008 was also the year in which another big thing happened: App Store (src: **https://en.wikipedia.org/wiki/App_Store_(iOS**).

Why does this matter to web development and offline at all? Availability.

The apps in the App Store had one big advantage: they would work when you were offline. Internet wasn't quite a thing, particularly mobile Internet and users of the App Store were mobile users!

For the years to come, users would be expecting and trusting that their apps would just work when they were unplugged from the web. That definitely set a bar.

Unfortunately, that wasn't the case for the case for the web. Websites were almost always considered second class citizens by almost all platforms they run in. The main reason behind it would have been security. The web proved to be quite a hostile world after all with the heap of ads that popped up from every corner of the world.

However, as time passed, people started building more and more complex applications on the web up to the point in which they weren't really websites anymore. What you are building with Redux today, we can guarantee you is not a just a website. You are storing data, talking to services, reacting to the user's input to render complex UIs, etc. In order for users to be able to rely on this apps in the same way they would with their desktop or mobile apps, they would have to be indeed available at all times.

It was clear that the web had to step up and at around 2010 the ApplicationCache API was born... And people absolutely hated it **http://alistapart.com/article/application-cache-is-a-douchebag**. Even though it sort of worked, it was very hard to make it do what you actually need and it would be very easy to mess things up. It was used in many places though and still is as its browser support is very good. Luckily for us the Chrome team sat to start tackling this problem and at around late 2014 we got the first version of Service Workers released. Its use is known as Progressive Web Apps. Application Cache is already deprecated and is currently in its way of being removed from the web platform (src: **https://html.spec.whatwg.org/multipage/browsers.html#offline**).

What made Service Workers different from Application Cache? They let the developer have absolute programmatic control over the network layer between the user and the web. This instantly translates into: our apps can now be available whenever the user needs them because they can work be taught to work offline.

The concept of Progressive Web Apps got so much traction over the last year that Google decided to run a Progressive Web Apps Summit just a couple of weeks before this book was first published, its keynote is a great summary of why it matters **https://www.youtube.com/watch?v=9Jef9IluQw0**.

Browser support in Service Workers is looking pretty good. As of the date of releasing this book, both Chrome and Firefox already have functional versions of it, Microsoft Edge is working on it and Safari said that they are considering it (src: **https://jakearchibald.github.io/isserviceworkerready/**).

This chapter is a bit different than the rest because it's not entirely Redux-specific. However, given the importance and the level of maturity web development is achieving, we thought it would be a good idea to present these concepts to our readers and show what the impact of such a technology is on your Redux application.

We will also touch on a side an often unexplored topic: offloading our store (and work in general) to Web Workers. Web workers have been around for a while but for one reason or another they haven't become mainstream in web development. They are totally a thing in native app development though, in which the main thread is a precious resource used to keep the UI running at 60fps+ and all the other processing happens in separate threads.

Using Service Workers in your app

A service worker (we will use the abbreviation SW from now on to keep it short and sweet), like all workers on the web, lives in its own separate file. The first thing we need to do in order to use it is to register it in your application.

```
// /app.js

// Because this is a new technology, many older browsers won't support it,
therefore it should be
// treated as an enhancement and we should check whether it's supported be-
fore trying to us it :).
if ('serviceWorker' in navigator) {
  // We then register our worker
  navigator.serviceWorker.register('/sw.js').then(reg => {
    // Here we can interact with it already!
    console.log('All set! :)', reg);
  }).catch(error => {
    // Oops, something went wrong
    console.log('Something went wrong :(...', error);
  });
}
```

The first thing you will notice is that a SW registration mechanism uses Promises. We've used them across the book but not in much depth; if you're not very familiar with them you can dive deeper into the concept here **http://www.html5rocks.com/en/tutorials/es6/promises**.

A couple of important things to take into account: SW only work through a web page served securely (over HTTPS) with a valid certificate. The origin of the script, i.e., the server it is hosted on, should be the same as the page that is registering it. And for the curious, yes, iframes can register their own SW too that will work within its scope. `localhost` is an exception to the rule and will work without HTTPS, which is handy to get started.

We will be working off the `offline` branch of our repository. You can test the example above with `npm start` and browsing to **http://localhost:8080**.

Once you `register` your worker, it will first be downloaded, then installed and finally activated.

As SW are completely asynchronous, you will deal with the last two through events.

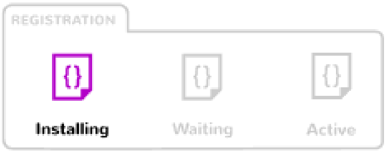
```
self.addEventListener('install', event => {
  // event.waitUntil takes a promise and is your hook to extend the installa-
tion process.
  event.waitUntil(
    doSomeWorkBeforeWeCanStart()
  );
});

self.addEventListener('activate', event => {
  // We're all set and ready to rock!
});
```

Here is the full SW's lifecycle according to the MDN:

Worker lifecycle

INSTALLING

This stage marks the beginning of registration. It's intended to allow to setup worker-specific resources such as offline caches.

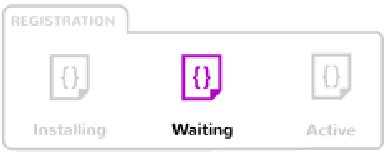


`install`

- Use **event.waitUntil()** passing a promise to extend the installing stage until the promise is resolved.
- Use **self.skipWaiting()** anytime before activation to skip installed stage and directly jump to activating stage without waiting for currently controlled clients to close.

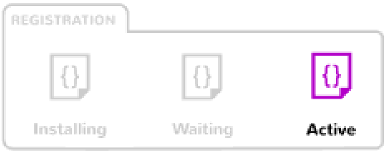
INSTALLED

The service worker has finished its setup and it's waiting for clients using other service workers to be closed.



ACTIVATING

There are no clients controlled by other workers. This stage is intended to allow the worker to finish the setup or clean other worker's related resources like removing old caches.

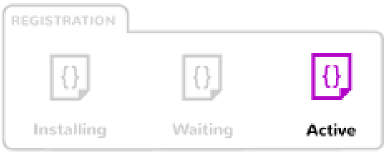


`activate`

- Use **event.waitUntil()** passing a promise to extend the activating stage until the promise is resolved.
- Use **self.clients.claim()** in the activate handler to start controlling all open clients without reloading them.

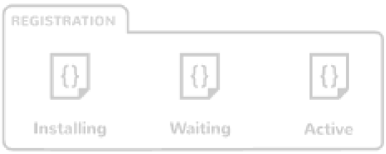
ACTIVATED

The service worker can now handle functional events.

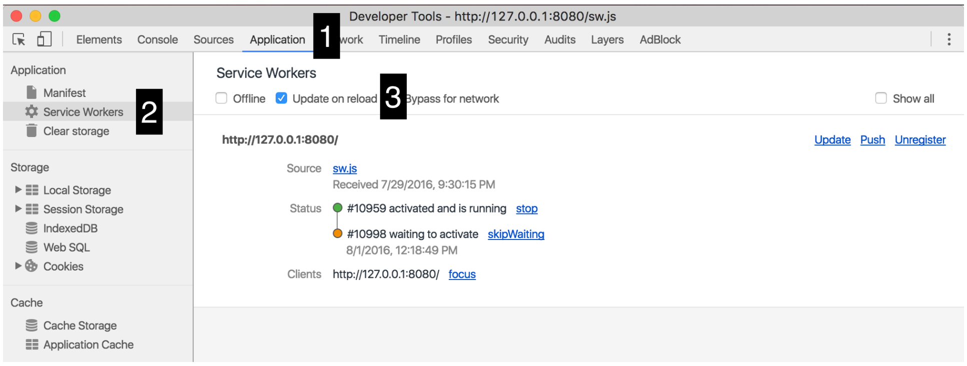


REDUNDANT

This service worker is being replaced by another one.



One important fact we should take away from the lifecycle explanation is that if we were to make changes to our SW and reload our page now, those changes might not be directly available. Luckily for us, the developer tools allow us to automatically reload the SW while on development mode, so before we dive any deeper and to avoid frustration, let's make sure we're reloading our SW every time we reload the page. You can activate this feature in Chrome by going to the Developer Tools, Application, Service Workers and selecting the checkbox for Update on reload. Here is a screenshot to make it easier to follow:



Service Workers

☐ Offline ☑ Update on reload ☐ Bypass for network ☐ Show all

http://127.0.0.1:8080/

Source sw.js
Received 7/29/2016, 9:30:15 PM

Status ● #10959 activated and is running stop
● #10998 waiting to activate skipWaiting
8/1/2016, 12:18:49 PM

Clients http://127.0.0.1:8080/ focus

You will notice that if you shutdown your server, your application isn't accessible anymore. We know what you might be wondering, didn't you promise that SW will take care of going offline? Yes! But we have to tell it how. It won't do anything until we instruct it to. Enter the `fetch` event.

```
self.addEventListener('fetch', event => {
  console.log(event.request);
});
```

The `fetch` event allows us to capture requests. You get `fetch` events for navigations within your SW's scope and any requests triggered by those pages, even if they're to another origin. This means you can intercept requests for all of your page's assets like CSS, JS, images, XHR, etc.

There are a few restrictions though:

- iframes and objects - they will use their own controller based on their resource URL.
- Service workers – requests to fetch/update a service worker don't go through the service worker.
- Requests triggered within a service worker – this is important to avoid loops.

The `event.request` is the request that triggered the event handler (src: **https://developer.mozilla.org/en-US/docs/Web/API/Request**). You can see all sort of stuff about it, like the URL, headers, etc. However, what makes the `fetch` event so attractive is that we can manipulate the request at will and even respond with another thing!

Implementing a middleware that can selectively store and sync data based on actions

WIP or pouch middleware. Initially based on pouch-redux-middleware, currently in use in UXtemple's Pages **https://github.com/UXtemple/usepages/blob/master/setup/configure-store.js#L25** to intercept actions **https://github.com/UXtemple/usepages/blob/master/setup/pouch-middleware-config.js** and sync (back and forth) with a CouchDB through PouchDB.

It can be abstracted away to use IDB without PouchDB although PouchDB itself is pretty light and covers most of the

```
import deepEqual from 'deep-equal';

export default function createPouchMiddleware(db, paths = []) {
  // create a whitelist of actions dispatched by redux that we should react
to
  const actionsWhitelist = paths
    .map(path => path.actionsWhitelist)
    .reduce((p, c) => p.concat(c), []);

  // cache documents to track changes from pouch to redux and vice-versa
  const cache = paths.map(() => ({}));

  // TODO merge differences with onNextState, this can already take care of
working against the
  // filter and we would have less loops, etc.
  function differences(docs, nextDocs) {
    const added = [];
    const updated = [];
    let removed = Object.keys(docs).map(id => docs[id]);

    Object.keys(nextDocs).forEach(id => {
      const nextDoc = nextDocs[id];

      // TODO optimise
      removed = removed.filter(doc => doc._id !== id);

      const doc = docs[id];
      if (!doc) {
        added.push(nextDoc);
      } else if (!deepEqual(doc, nextDoc)) {
        updated.push(nextDoc);
      }
    });

    return {
      added,
      updated,
      removed
    };
  }

  function load(path, i, nextState) {
    cache[i] = {
      ...cache[i],
      ...path.get(nextState)
    };
  }


  function onNextState(path, i, nextState) {
    const { added, removed, updated } = differences(cache[i], path.get(next-
State));

    const onNextStateAddOrUpdate = doc => cache[i][doc._id] = doc;
    const onNextStateRemove = doc => {
      delete cache[i][doc._id];
      return { ...doc, _deleted: true };
    };

    const diff = added.map(onNextStateAddOrUpdate)
      .concat(removed.map(onNextStateRemove), updated.map(onNextStateAddOrUp-
date));

    // TODO handle results somehow?
    db.bulkDocs(diff);
  }

  function setup(dispatch) {
    // subscribe new changes in pouch
    const changes = db.changes({
      include_docs: true,
      live: true,
      since: 'now'
    });

    function onDbChange({ doc }) {
      paths.forEach((path, i) => {
        // see which paths are affected by the change
        if (path.match(doc)) {
          let action;
          // find the old doc
          const oldDoc = cache[i][doc._id];
          // the doc might have been removed on some other place
          if (doc._deleted) {
            // and we may have it locally
            if (oldDoc) {
              // if so, delete it from our cache
              delete cache[i][doc._id];
              // and instruct the redux store to remove it too
              action = path.actions.remove(doc);
            }
          } else {
            // cache the document reference
            cache[i][doc._id] = doc;

            // if the doc existed, trigger an update, otherwise add it to
the redux store
            action = oldDoc ? path.actions.update(doc) : path.ac-
tions.add(doc);
          }

          if (action) {
            dispatch(action);
          }
        }
      });
    }

    // react to those remote changes
    changes.on('change', onDbChange);

    return changes;
  }

  return store => {
    setup(store.dispatch);

    return next => action => {
      const returnValue = next(action);

      if (actionsWhitelist.indexOf(action.type) !== -1) {
        const nextState = store.getState();

        paths.forEach((path, i) => {
          if (path.actionLoad === action.type) {
            load(path, i, nextState);
          } else {
            onNextState(path, i, nextState);
          }
        });
      }
      return returnValue;
    };
  };
}
```

Offloading the store’s work to workers

https://github.com/chikeichan/redux-worker: Redux-Worker helps you use web workers in your Redux applications. It does so by enhancing your Redux store with:

- A dispatcher that posts messages to a worker thread.
- A replacement reducer that listens to messages from a Web Worker and then updates state.
- A mechanism to register and execute tasks in a Web Worker using the dispatcher, which returns a promise that will resolve when the task completes. It spawns just one Web Worker instance, so don’t give it more credit than it deserves. It does not provide an interface for spawning multiple workers and balancing work between them: this is a much more complicated problem that Redux-Worker is not attempting to solve.

Here are more specific use cases like usepages compiler. WIP the watcher (will probably change in pages itself but it’s ok as an example for now) dispatches a run action **https://github.com/UXtemple/usepages/blob/master/compiler/watch.js#L11**

```
import { run } from './actions';
import debounce from 'lodash.debounce';

// TODO this can be worked the other way around, we don't need a watcher on
this end,
// the worker can tell when there's new code
export default function watch(store) {
  const watcher = async () => {
    const state = store.getState();

    if (!state.compiler.isRunning && state.compiler.shouldRun) {
      store.dispatch(run());
    }
  };

  return store.subscribe(debounce(watcher, 150));
}
```

That in turn posts a message to the worker to “transform” the code **https://github.com/UXtemple/usepages/blob/master/compiler/compile.js#L10**

```
export default function compile(blocks = [], pages = []) {
  // ...
  // transform the source
  const code = await worker.postMessage({
    type: 'transform',
    blocksSource,
    pagesSource
  });

  // ...
}
```

And then the worker "routes" accordingly:

```
import * as compiler from '../compiler/transform';
import register from 'promise-worker/register';

register(async message => {
  switch (message.type) {
    // ...

    case 'transform':
      return await compiler.transform(message.blocksSource, message.pages-
Source);
      break;

    default: break;
  }
});
```

Recommended reads

If you want to dive deeper in the subject, we totally recommend you to read the following:

- Offline Web Applications **https://www.udacity.com/course/offline-web-applications--ud899**. An amazing, step by step Udacity course by Jake Archibald in which he seeks to show developers the impact and importance that going offline has in UX. He also dives into IndexedDB. Jake's blog **https://jakearchibald.com** also has tons of content related to Service Workers. He's a Google Chrome developer advocate and has been vouching for Service Workers since day one.
- Pokedex: **https://www.pokedex.org**. An offline-first application built by Nolan Lawson to showcase how we can leverage a variety of workers to make our apps be indistinguishable from native apps. Nolan is one of the brains behind PouchDB, the offline-first database.
- Beyond Progressive Web Apps (part 1 **http://hood.ie/blog/beyond-progressive-web-apps-part-1.html**, part two **http://hood.ie/blog/beyond-progressive-web-apps-part-2.html** and part 3 **http://hood.ie/blog/beyond-progressive-web-apps-**

part-3.html). The HOODIE team has been doing amazing work on the offline-first front since their conception and Jan Lehnardt does a brilliant job at going beyond the surface of Service Workers, the challenges you face with syncing, etc. Being one of the main contributors to CouchDB he clearly knows what he's talking about :).

- Service workers explained **https://github.com/slightlyoff/ServiceWorker/blob/master/explainer.md**. A very comprehensive and succinct Service Workers FAQ by Alex Russell who works on the Chrome team.
- Progressive Web Apps **https://developers.google.com/web/progressive-web-apps**. Google's own set of resources on the topic. Lots of tutorials and code samples here.
- Service Worker Cookbook **https://serviceworke.rs**. Mozilla's collection of working, practical examples of using service workers in modern web apps.
- Service workers, without the work **https://developers.google.com/web/tools/service-worker-libraries/?hl=en**. A Google initiative to reduce the boilerplate needed to run service workers with sensible defaults.
- Promise workers **https://github.com/nolanlawson/promise-worker**. Nolan Lawson's take at promisify-ing workers. It works amazingly well on both, web and service workers, and has a very low footprint.

Conclusion

We hope you have learned as much about Redux as we had writing this book and creating the sample application.