Redux IN ACTION

Marc Garreau
Will Faurot

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Redux in Action

Version 4

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP edition of *Redux in Action*.

A fair question to ask is whether a full-sized book is even necessary for a library as small as Redux. In our earliest conversations with the Manning team, we were uncertain. After reflecting on our extensive experience mentoring newcomers to React and Redux, we agreed that learning the fundamentals of Redux is not all that difficult. You can expect to accomplish that level of mastery within the first couple chapters of this book.

When ready to move beyond the basics however, we find that developers commonly hit a wall. Redux enables complex state management in an application, but it leaves a great many usage decisions up to the developer. At this point, beginners are typically eager to learn where to put certain logic, which components to connect to the Redux store, how to handle side effects, and other best practices. These best practices lay scattered across documentation, blog posts, conference talks, isolated chapters of books, series of tweets, and the like.

Beyond best practices, there is also a vast ecosystem of tools that play nicely with Redux. React and Redux leave plenty of configurable holes in an application that can be filled by things like middleware or store enhancers. Many tutorials fall prey to introducing a confusing number of these additive tools, conflating several ideas in the learner’s mind. One of our primary goals with this full-length title is to introduce real-world complexity around a Redux application one manageable layer at a time.

In order to quickly get into the nitty-gritty, we assume our readers have some experience under their belts. Proficiency with JavaScript, including the best-known features of ECMAScript 2015, is a prerequisite. Additionally, we expect familiarity, but not expertise, with React. For those totally unfamiliar with React, there are a couple of excellent titles in the Manning library you may wish to visit first.

While in MEAP, this book is very much a living, breathing, work in progress. Whether you’re brand new to Redux or have been around the block, we’re eager to hear to your feedback as you read. Please feel free to let us know your thoughts in the Author Online forum, or reach out to us directly on social media. Publishing is a bucket list item for us both and we’re grateful for your help along the way. Thanks again for your interest and happy reading!

—Marc Garreau
Will Faurot

brief contents

1 Introducing Redux

This chapter covers:

- **What Redux is**
- **The differences between Flux and Redux**
- **How Redux works with React**
- **An introduction to actions, reducers, and the store**
- **When you should use Redux**

If you hop into any React web application in 2017, there's a good chance you'll find Redux there to manage its state. It's remarkable that we reached this place so quickly, though. Just a few years ago, Redux had yet to be created and React enjoyed an excited and blossoming user base. Early adopters of React agreed that they had found the best solution yet to the view layer — the "V" of the MVC front-end framework puzzle. What they couldn't agree on was how to manage the state of those applications once they became the size and complexity that the real-world demands.

1.1 What is state?

React components have the concept of local, or component state. Within any given component, you can keep track of the value of an input field or whether a button has been toggled, for example. Local state makes easy work of managing a single component's behavior. However, today's single page applications often require synchronizing a complex web of state. Nested levels of components may render a different user experience based on the pages a user has already visited, the status of an AJAX request, or whether a user is logged in.

Let's consider a use case involving the authentication status of a user. Your product manager tells you that when a user is logged into an ecommerce store, the navigation bar should display the user's avatar image, the store should display items nearest to the user's zip code first, and the newsletter signup form should be hidden. Within a vanilla React architecture, your options are limited for syncing state across each of the components. In the end, you'll likely end up passing the authentication status and additional user data from one top-level component down to each of these nested components.

This architecture has several disadvantages. Along the way, data may filter through components that have no use for it other than to pass the data on to their children. In a large application, this can result in tons of data moving through unrelated components, passed down via props or passed up via callbacks. It's likely that a small number of components at the top of the application end up with an awareness of most of the state used throughout the entire application. At a certain scale, maintaining and testing this code becomes untenable. Because React was not intended to solve the same breadth of problems that other MVC frameworks attempted to address, an opportunity existed to bridge those gaps.

With React in mind, Facebook eventually introduced Flux, an architecture pattern for web applications. Flux became tremendously influential in the world of front-end development and began a shift in how we thought about state management in client-side applications. Facebook offered its own implementation of this pattern, but soon more than a dozen "Flux-inspired" state management libraries emerged and competed for React developers' attention.

This was a tumultuous time for React developers looking to scale an application. We saw the light with Flux, but continued to experiment to find more elegant ways to manage complex state in applications. For a time, newcomers encountered a paradox of choice; a divided community effort had produced so many options, it was anxiety-inducing. To our surprise and delight though, the dust is already settling and Redux has emerged as a clear winner.

Redux took the React world by storm with a simple premise, a big payoff, and a memorable introduction. The *premise* is to store your entire application state in a single object using pure functions. The *payoff* is totally predictable application state. The *introduction*, for most early users, came in Dan Abramov's 2015 React Europe conference talk, titled "Live React: Hot Reloading with Time Travel." Dan wowed attendees by demonstrating a Redux developer experience that blew established workflows out of the water. A technique called "hot loading" made live application updates while maintaining existing state, and his nascent Redux developer tools enabled you to "time travel" through application state — rewinding and replaying user actions with a single click. The combined effect offers developers debugging super powers, which we'll explain in detail in chapter 3.

To understand Redux, we'd first like to properly introduce you to Flux, the architecture pattern developed at Facebook and credited to Jing Chen. Redux and many of its alternatives are variations of this Flux architecture.

1.2 What is Flux?

Flux is foremost an architecture pattern. It was developed as an alternative to the prevailing Model-View-Controller (MVC) JavaScript patterns popularized by incumbent frameworks, such as Backbone, Angular, or Ember. Although each framework puts its own spin on the MVC pattern, many share similar frustrations: generally, the flow of data between models, views and controllers can be difficult to follow.

Many of these frameworks use two-way data binding, in which changes to the views update corresponding models, and changes in the models update corresponding views. When any given view can update one or more models, which in turn can update more models, you can't be blamed for losing track of the expected outcome at a certain scale. Chen contested that although MVC frameworks work well for smaller applications, the two-way data-binding models that many of them employ don't scale well enough for the size of Facebook's application. Developers at the company became apprehensive of making changes, for fear of the tangled web of dependencies producing unintended consequences.

Flux sought to address the unpredictability of state and the fragility of a tightly coupled model and view architecture. Chen went about scrapping the two-way data binding model in favor of a unidirectional data flow. Instead of permitting each view to interact with their corresponding models, Flux requires all changes to state to follow a single path. When a user clicks a Submit button on a form, for example, an action is sent to the application's one and only dispatcher. The dispatcher will then send the data through to the appropriate data stores for updating. Once updated, the views will become aware of the new data to render. Figure 1.1 illustrates this unidirectional data flow.

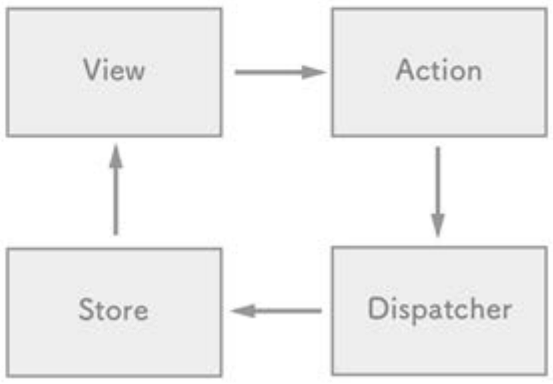


Figure 1.1 Flux specifies that data must flow in a single direction.

1.2.1 Actions

Every change to state starts with an *action* (figure 1.1). An action is an object describing an event in your application. They are typically generated by either a user interaction or by a server event, like an HTTP response.

1.2.2 Dispatcher

All data flow in a Flux application is funneled through a single *dispatcher*. The dispatcher itself has very little functionality, because its purpose is to receive all actions and send them to each store that has been registered. Every action will be sent to every *store*.

1.2.3 Stores

Each *store* manages the state of one domain within an application. In an ecommerce site, you may expect to find a shopping cart store and a product store, for example. Once a store is registered with the dispatcher, it will begin to receive actions. When it receives an action type that it cares about, the store will update accordingly. Once a change to the store is made, an event is broadcast to let the views know to update using the new state.

1.2.4 Views

Flux may have been designed with React in mind, but the *views* aren't required to be React components. For their part, the views need only subscribe to the stores they wish to display data from. The Flux documentation encourages the use of the controller-view pattern, whereby a top-level component handles communication with the stores and passes data to child components. Having both a parent and a nested child component communicating with stores can lead to extra renders and unintended side-effects.

Again, Flux is an architecture pattern first. The Facebook team maintains one simple implementation of this pattern, aptly (or confusingly, depending on your perspective) named Flux. Many alternative implementations have emerged since 2014, including Alt, Reflux, and Redux. A more comprehensive list of those alternative implementations can be found in section 1.4.

1.3 What is Redux?

We can't put it much better than the official docs: "Redux is a predictable state container for JavaScript applications." It's a standalone library, but is used most often as a state management layer with React. Like Flux, its major goal is to bring consistency and predictability to the data in our applications. Redux divides the responsibilities of state management into a few separate units:

- The store holds all of your application state in a single object. (We'll commonly refer to this object as the *state tree*.)
- The store can be updated only via actions, an object describing an event.
- Functions known as *reducers* specify how to transform application state. *Reducers* are functions that take the current state in the store and an action, and return the next

state after applying any updates.

Technically speaking, Redux may not qualify as a Flux implementation. It nontrivially deviates from some of the components of the prescribed Flux architecture, such as the removal of the dispatcher altogether. Ultimately though, Redux is very Flux-like and the distinction is a matter of semantics.

Redux enjoys the benefits of a predictable data flow from the Flux architecture, but also found ways to alleviate the uncertainty of store callback registrations. As alluded to in the previous section, it can be a pain to reconcile the state of multiple Flux stores. Redux, instead, prescribes a single store to manage the state of an entire application. We'll learn more about how this works and what the implications are in the coming sections.

1.3.1 React and Redux

Although Redux was designed and developed in the context of React, the two libraries are completely decoupled. React and Redux are connected via bindings, shown simply in figure 1.2.

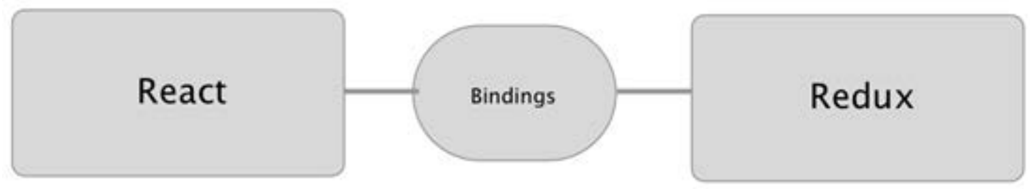


Figure 1.2 Redux isn't part of any existing framework or library, but additional tools called *bindings* connect Redux with React. Over the course of the book we'll use the react-redux package for this.

It turns out that the Redux paradigm for state management can be implemented alongside most JavaScript frameworks. Bindings exist for Angular 1 and 2, Backbone, Ember, and many more technologies.

Although this book is fundamentally about Redux, our treatment of it is closely tied with React. Redux is small standalone library, but it fits particularly well with React components. Redux will help you define what your application does; React will handle how your application looks.

Most of the code we'll write over the course of the book, not to mention most of the React/Redux code you'll write period, will fall into a few categories:

- The application's state and behavior, handled by Redux.
- Bindings, provided by the react-redux-package, that connect the data in the Redux store with the view (React components).
- Stateless components that comprise much of your view layer.

You'll find that React is a very natural ecosystem for Redux. While React has mechanisms to manage state directly in components, the door is wide open for Redux to come in and manage the greater application state. If you're interested in an alternative ecosystem, chapter 12 explores the relationship between Redux and some of the other JavaScript frameworks.

1.3.2 The Three Principles

You will have covered substantial ground by grokking that state in Redux is represented by a single source of truth, is read-only, and changes to it must be made with pure functions.

SINGLE SOURCE OF TRUTH

Unlike the various domain stores prescribed by the Flux architecture, Redux manages an entire application's state in one object, inside of one store. The use of a single store has important implications. The ability to represent the entire application state in a single object simplifies the developer experience; it becomes dramatically easier to think through the application flow, predict the outcome of new actions, and debug issues produced by any given action. The potential for *time travel* debugging, or the ability flip back and forth through snapshots of application state, is what inspired the creation of Redux in the first place.

STATE IS READ-ONLY

Just like Flux, actions are the only way to initiate changes in application state. No stray AJAX call can produce a change in state without being communicated via an action. Redux differs from many Flux implementations though, in that these actions do not result in a mutation of the data in the store. Instead, each action results in a shiny new instance of the state to replace the current one. More on that subject in the next bullet point.

CHANGES ARE MADE WITH PURE FUNCTIONS

Actions are received by reducers. It's important that these reducers be pure functions. Pure functions are deterministic; they always produce the same output given the same inputs, and they don't mutate any data in the process. If a reducer were to mutate the existing state while producing the new one, we may end up with erroneous new state, but we also lose the predictable transaction log that each new action is intended to provide. The Redux developer tools and other features, such as undo and redo functionality, rely on application state being computed by pure functions.

1.3.3 The Workflow

So far, we've touch briefly upon things like actions, reducers, and the store, but in this section, we'll cover each in more depth. What's important to take away here is the role that each element plays, and how they work together to produce a desired result. For now, don't worry about finer implementation details, because we'll have plenty of time in later chapters to apply the concepts we're about to explore.

Modern web applications are ultimately about handling events. They could be initiated by a user, like navigating to a new page or submitting a form. Or they could be initiated by another external source, like a server response. Responding to events usually involves updating state and re-rendering with that updated state. The more that your application does, the more state you need to track and update. Combine this with the fact that most of these events occur asynchronously, and you suddenly have some real obstacles to maintaining an application at scale.

Redux exists to create structure around how you handle events and manage state in your application, hopefully making you a more productive and happy human in the process.

Let's look at how we might handle a single event in an application using Redux and React. Say you were tasked with implementing one of the core features of a social network, adding a post to your activity feed. Here's a quick mockup of a user profile page, which may or may not take its inspiration from Twitter.

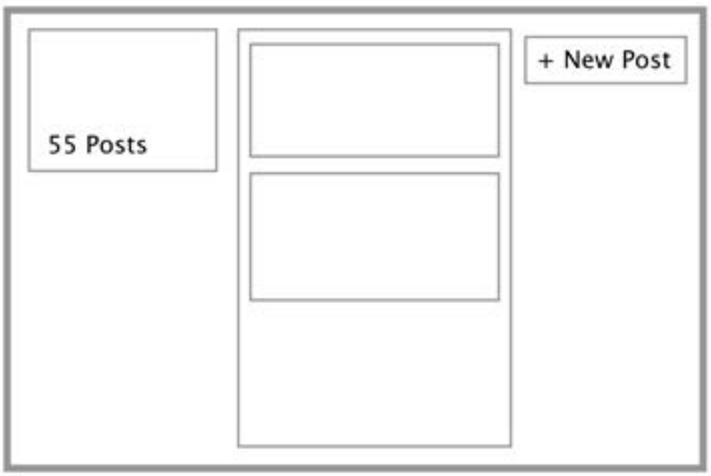


Figure 1.3 A simple mockup of a profile page. This page is backed by two main pieces of data: the total post count, and the list of post objects in the user's activity feed.

The following distinct steps are involved in handling an event like a new post:

- From the view, indicate that an event has occurred (a post submission) and pass along the necessary data (the content of the post to be created).
- Update state based on the type of event — add an item to the user's activity feed and increment the post count.
- Re-render the view to reflect the updated state.

Sounds reasonable, right? If you've used React before, you've likely implemented features like this directly in components. Redux takes a different approach. Code to satisfy the three tasks is moved out of React components into a few separate entities. We're already familiar with the View in figure 1.4, but we're excited to introduce a new cast of characters you'll hopefully learn to love.

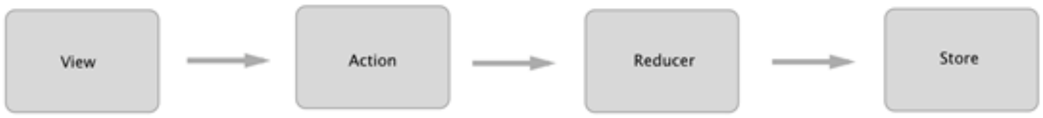


Figure 1.4 A look at how data flows through a React/Redux application. We've omitted a few common pieces like middleware and selectors, which we'll cover in-depth in later chapters.

ACTIONS

We want to do two things in response to the user submitting a new post: add the post to the user's activity feed and increment their total post count. After the user submits, we'll kick off the process by dispatching an action. Actions are plain old JavaScript objects that represent an event in your application, as follows:

```
{
  type: 'CREATE_POST',
  payload: {
    body: 'All that is gold does not glitter'
  }
}
```

Let's break that down. We have an object with two properties:

- type — a string that represents the category of action being performed. By convention, this property is capitalized and uses underscores as delimiters.
- payload — an object that provides the data necessary to perform the action. In our case, we only need one field: the contents of the message we want to post.

Actions have the advantage of serving as audits. There's a historical record of everything happening in your application, including any data needed to complete a transaction. It's hard to understate how valuable this is in maintaining a grasp on a complex application. Once you get used to having a highly readable stream describing the behavior of your application in real-time, you'll find it hard to live without.

Throughout the book, we'll frequently come back to this idea of *what* versus *how*. You can think of Redux as decoupling *what* happens in an application from *how* we respond to an event. Actions handle the *what* in this equation. They simply describe an event, they don't know and don't care what happens downstream. Somewhere down the road we'll eventually have to specify *how* to handle an action. Sounds like a job fit for a reducer!

REDUCERS

Reducers are functions responsible for updating your state in response to actions. They are simple functions that take your current state and an action as arguments, and return the next state.

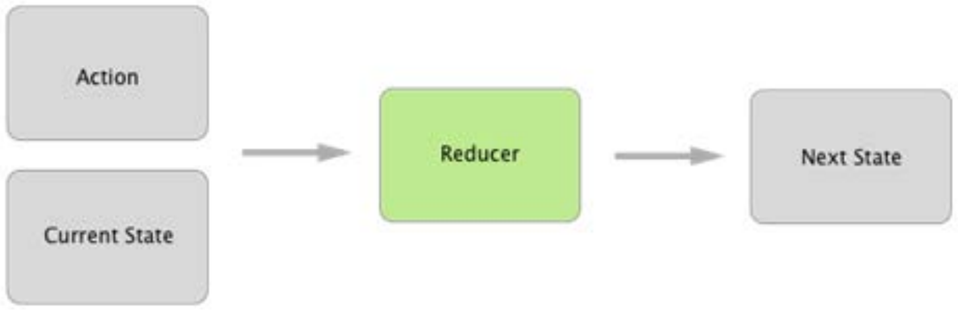
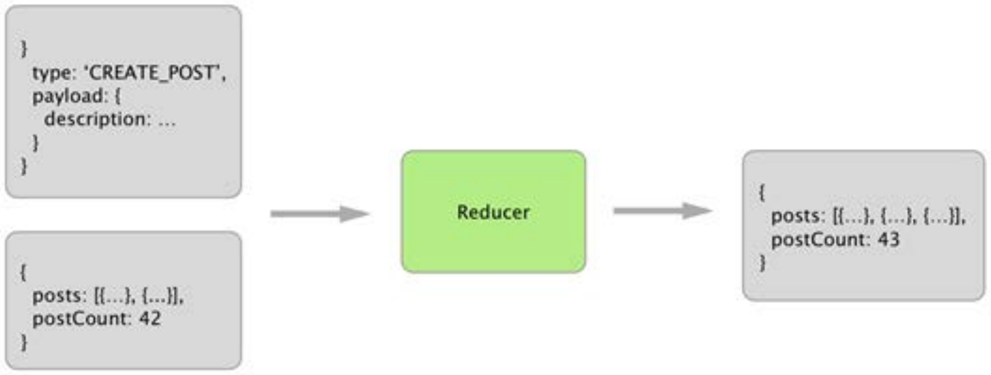


Figure 1.5 An abstract representation of a reducer's function signature. If this diagram looks simple, that's because it is. Reducers are meant to be simple functions that compute a result, making them easy to work with and test.

Reducers are typically very easy to work with. Just like all pure functions, they produce no side-effects. They don't affect the outside world in any way, and they're referentially transparent. The same inputs will always yield the same return value. This makes them particularly easy to test. Given certain inputs, you can verify that you receive the expected result. Here's how our reducer might update the list of posts and the total post count:


```
}
  type: 'CREATE_POST',
  payload: {
    description: ...
  }
}
```

```
{
  posts: [{...}, {...}],
  postCount: 42
}
```

```
{
  posts: [{...}, {...}, {...}],
  postCount: 43
}
```

Figure 1.6 Visualizing a reducer hard at work. It accepts as input an action and the current state. The reducer simply must specify what the new state should look like. No mutations, no side-effects, no funny business. Data in, data out.

We're focusing on a single event in this example, which means we need only one reducer. However, you certainly aren't limited to only one. In fact, more sizable applications frequently implement several reducer functions, each concerned with a different slice of the state tree. These reducers are combined, or composed, into a single "root reducer."

STORE

Reducers describe how to update state in response to an action, but can't modify state directly. That privilege rests solely with the store.

In Redux, application state is stored in a single object. The store has a few main roles, which follow:

- Hold application state.
- Provide a way to access state.
- Provide a way to specify updates to state. The store requires an action be dispatched to modify state.
- Allow other entities to subscribe to updates (React components in our case). View bindings provided by react-redux will allow us to receive updates from the store and respond to them in our components.

The reducer processed the action and computed the next state. Now it's time for the store to update itself and broadcast the new state to all registered listeners (we care specifically about the components that make up our profile page).

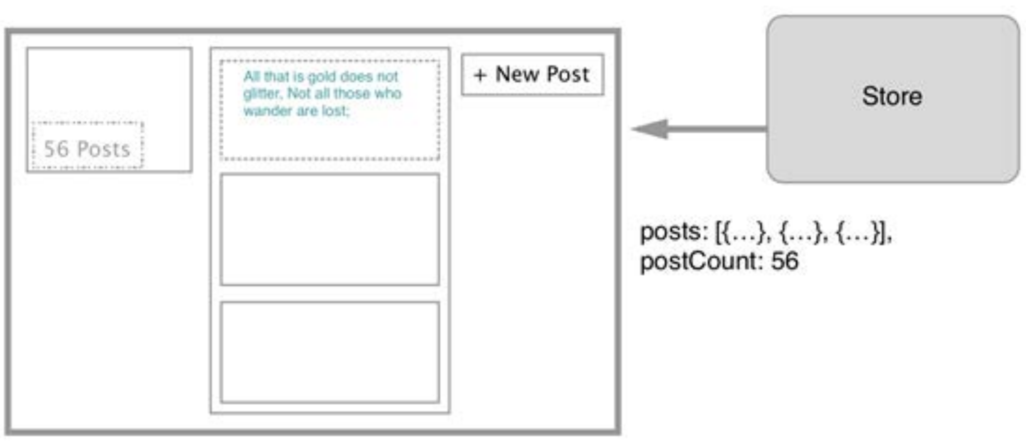

56 Posts
All that is gold does not glitter. Not all those who wander are lost;
+ New Post
Store
posts: [{…}, {…}, {…}],
postCount: 56

Figure 1.7 The store now completes the loop by providing the new state to our profile page. Notice that the post count has incremented, and the new post has been added to the activity feed. If our user adds another post, we'd follow the same exact flow. The view would dispatch an action, reducers specify how to update state, and the store broadcasts the new state back to the view.

Now that you're familiar with some of the most important building blocks, let's take a look at a more comprehensive diagram of the Redux architecture. Some of pieces will be unfamiliar now, but we will revisit this diagram repeatedly throughout this book, and over time, we'll fill in each of those gaps.

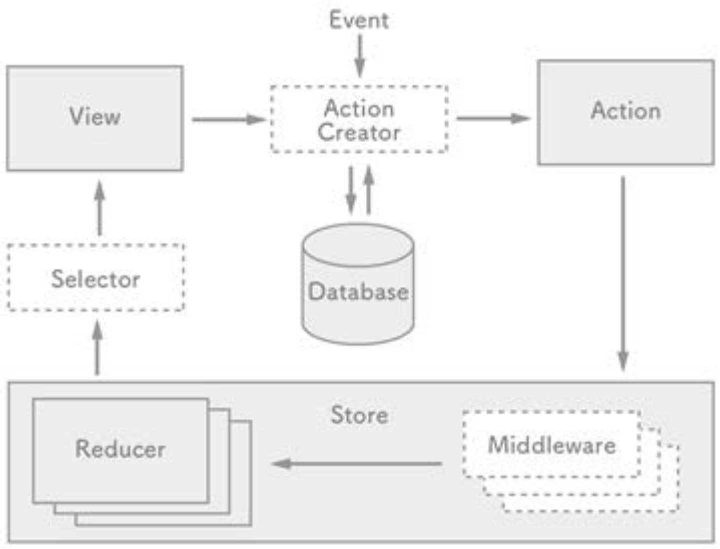


Figure 1.8 This diagram will serve to anchor our understanding of the elements of Redux as we move forward. So far, we've talked about actions, reducers, the store, and views.

To review, an interaction with a view may produce an action. That action will filter through one or more reducers and produce a new state tree within the store. Once the state has updated, the views will be made aware that there is new data to render. That's the whole cycle! Items in this diagram with a dashed border (action creators, middleware, and selectors) are optional, but powerful tools in a Redux architecture. We will cover each of these topics in future chapters.

If this feels like a lot, don't fret. If you're new to the kind of one-directional architecture that we're beginning to explore, it can be initially overwhelming (we certainly thought so at first). It takes time to let these concepts sink in. Developing a sense for what role they play and what type of code belongs where is as much art as it is science, and it's a skill you'll develop over time as you continue get your hands dirty.

1.4 Why Should I Use Redux?

By this point, you've been exposed to many of the Redux talking points. Just in case you have to pitch your boss on Redux by the time you finish the first chapter, let's consolidate those ideas into a highlight reel. In short, Redux is a small, easy-to-learn state management library that results in a highly predictable, testable, and debuggable application.

1.4.1 Predictability

The biggest selling point for Redux is the sanity it provides to applications juggling complex state. The Redux architecture offers a straightforward way to conceptualize and manage state, one action at a time. Regardless of application size, actions within the unidirectional data flow result in predictable changes to a single store.

1.4.2 Developer Experience

Predictability enables some world-class debugging tools. Hot-loading and time-travel debugging provide developers with wildly faster development cycles, whether building new features or hunting down bugs. Your boss will like that you're a happier developer, but she'll love that you're a faster one.

1.4.3 Testability

The Redux implementation code you'll write is primarily functions, many of them pure. Each piece of the puzzle can be broken out and unit-tested in isolation with ease. Official documentation makes use of Jest and Enzyme, but whichever JavaScript testing libraries your organization prefers will do the trick.

1.4.4 Learning Curve

Redux is a pretty natural step up from vanilla React. The library has a remarkably small footprint, exposing only a handful of APIs to get the job done. You can become familiar with all of it in a day. Writing Redux code also requires your team to become familiar with some functional programming patterns. This will be new territory for some developers, but the concepts are straightforward. Once you understand that changes to state can be produced only by pure functions, you're most of the way there.

1.4.5 Size

If your boss is doing her job, one of the items on her checklist is dependency size. Redux is a tiny library, just under 7KB when minified. Checkmate.

1.5 When Should I Use Redux?

There's no question we've been hitting you over the head with how great Redux is, but it's certainly no cure-all. We've argued in favor of why you should use Redux, but as we all know, nothing in life is free and no software pattern exists without tradeoffs.

The cost of Redux is a fair amount of boilerplate code and the added complexity of something more than React's local component state. It's important to realize that Redux, and the usage patterns you establish while using it, is one more thing for a new developer on your team to learn before they can contribute.

Redux co-creator Dan Abramov weighed in here, even publishing a blog post entitled "You Might Not Need Redux." He recommends starting without Redux and introducing the library only after you've reached enough state management pain points to justify including it. The recommendation is intentionally vague, because that turning point will be slightly different for every team. Smaller applications without complex data requirements are the most common scenario where it might be more appropriate to not use Redux in favor of plain React.

What might those pain points look like? There are a few common scenarios that teams use to justify bringing in Redux. The first is the passing of data through several layers of components that don't have any use for it. The second scenario deals with sharing and syncing data between unrelated parts of the application. We all have a tolerance for doing some of the above in React, but eventually there comes a breaking point.

Redux is likely a good fit out of the gate if you know you'll want to build a specific feature that it excels at. If you know your application will have complex state and require undo and redo functionality, cut to the chase and pull in Redux. If server-side rendering is a requirement, consider Redux up front.

1.6 Alternatives to Redux

As mentioned already, Redux entered a crowded state-management market and more options have appeared since. Let's run through the most popular alternatives for managing state in React applications.

1.6.1 Flux Implementations

While researching, we stopped counting Flux implementation libraries somewhere in the low 20s. Astoundingly, at least eight of them have received more than 1000 stars on GitHub. This highlights an important era in React's history; the Flux architecture was a groundbreaking idea that spurred a lot of excitement in the community and, as a result, a great deal of experimentation and growth. During this period, libraries came and went at such an exhausting rate that the term *JavaScript Fatigue* was coined. With hindsight, it's clear that each of those experiments was an important stepping stone along the way. Over time, many of the alternative Flux implementation maintainers have graciously bowed out of the race in favor of Redux or one of the other popular options, but there are still several well-maintained options out there.

FLUX

Flux, of course, is the one that started it all. In the maintainers' own words, "Flux is more of a pattern than a framework." A lot of great documentation about the Flux architecture pattern lives in this repository, but a small API is exposed to facilitate building applications with the architecture. The Dispatcher is at the core of that API, and in fact, several other Flux implementations have incorporated that Dispatcher into their libraries. Measured in GitHub stars, this library is about half as popular as Redux and continues to be actively maintained by the Facebook team.

REFLUX

Reflux was a fast follow to the original Flux library. The library introduces some functional reactive programming ideas to the Flux architecture by ripping out the single Dispatcher in favor of giving each action the ability to dispatch themselves. Callbacks can be registered with actions to update stores. Reflux is still maintained and about one sixth as popular as Redux, measured by GitHub stars.

ALT

Unlike Reflux, Alt stays true to the original Flux ideas and makes use of the Flux Dispatcher. Alt's selling points are its adherence to the Flux architecture and a reduction in boilerplate code. Although it once enjoyed an enthusiastic community, at the time of writing, there have been no commits to the project in more than six months.

HONORABLE MENTIONS

To round out the bunch with greater than 1000 GitHub stars, we also have Fluxible, Fluxxor, NuclearJS, and Flummox. Fluxible continues to be well-maintained by the Yahoo team. Fluxxor, NuclearJS and Flummox may be maintained, but are no longer active. To underscore the idea that these projects were important stepping stones, Flummox was created by Andrew Clark, who went on to co-create Redux with Dan Abramov.

1.6.2 MobX

MobX offers a functional reactive solution to state management. Like Flux, MobX uses actions to modify state, but components react to that mutated, or observable, state. Although some of the terminology in functional reactive programming can be intimidating, the features are pretty approachable in practice. MobX also requires less boilerplate code than Redux, but does more for you under the hood and is therefore less explicit. The first commits for MobX predate those of Redux by only a couple of months, in early 2015.

1.6.3 GraphQL Clients

GraphQL is an exciting new technology, also being developed by the Facebook team. It's a query language that allows you to specify and receive exactly the data that is required by a component. This paradigm fits well with the intended modularity of React components; any data fetching that is required by the component is encapsulated within it. Queries to the API are optimized for the data needs of parent and children components.

Typically, GraphQL is used with a GraphQL client. The two most popular clients today are Relay and Apollo Client. Relay is another project developed and maintained by the Facebook team (and open-source community). Apollo is implemented with Redux under the hood.

While some developers will be so bold as to bring in both Redux and a GraphQL client to manage the same application's state, we find this to be overly complex and generally unnecessary. They solve many of the same problems and either one will typically do the trick on their own.

1.7 Summary

This chapter introduced the Flux architecture pattern and where Redux ran with those ideas. You learned several practical details about the library, including the following:

- Redux state is stored in a single object and is the product of pure functions.
- For the price of some boilerplate code, Redux can introduce predictability, testability, and debuggability to your complex application.
- If you’re experiencing pain points while syncing state across your application or passing data through multiple component layers, consider introducing Redux.

Now you’re ready to put the basic building blocks together and see a functioning Redux application, end to end. In the next chapter, we’ll build a task management application with React and Redux.

2 Your first Redux application

This chapter covers

- **Configuring a Redux store**
- **Connecting Redux to React with the react-redux package**
- **Actions and action creators**
- **Using reducers to update state**
- **Container and presentational React components**

By now, you're almost certainly itching to get started on a Redux application. You have more than enough context to begin, so let's scratch that itch. This chapter will guide you through the set up and development of a simple task-management application leveraging Redux to manage its state.

By the end of the chapter, we'll have walked through a complete application, but more importantly, you will have learned enough of the fundamentals to leave the nest and create simple Redux applications of your own. Through the introduction of some components that were strategically omitted in chapter 1, you'll develop a better understanding of the unidirectional data flow and how each piece of the puzzle contributes to that flow.

You may wonder if introducing Redux is overkill for the small application we'll be building in this chapter. To iterate a point made in chapter 1, we encourage the use of vanilla React up until you experience enough pain points to justify bringing Redux in.

If this chapter were the whole of it, Redux would indeed be overkill. It's not until we reach features introduced in later chapters that it really begins to make sense. As a matter of practicality, we'll head straight for Redux; that's why we're here, after all! As a thought experiment, you may enjoy rebuilding the application in React to determine when including Redux makes sense for yourself, once you become comfortable with the fundamentals.

2.1 Building a Task Management Application

The path we'll walk is a well-trodden one: building a project task-management application. In this chapter, we'll implement some simple functionality, but we'll add increasingly complex features throughout the rest of the book as we cover each concept in more detail.

Our app is lovingly named *Parsnip*. Why Parsnip? No good reason. It spoke to us in the moment, and we went with it. Specifically, Parsnip will be a Kanban board, a tool that allows users to organize and prioritize work (similar to Trello, Waffle, Asana, and a number of other tools). An app like this is highly interactive and requires complex state management — a perfect vehicle for us to apply our Redux skills.

To see Redux in action without a lot of bells and whistles, we'll start with one resource, a *task*. Our users should be able to:

- Create a new task with three properties: *title*, *description*, and *status*
- See a list of all tasks
- Change the *status* of a task. For example, a task may move from Unstarted, to In Progress, and finally to Completed

By the end of the chapter, we'll have something like Figure 2.1.

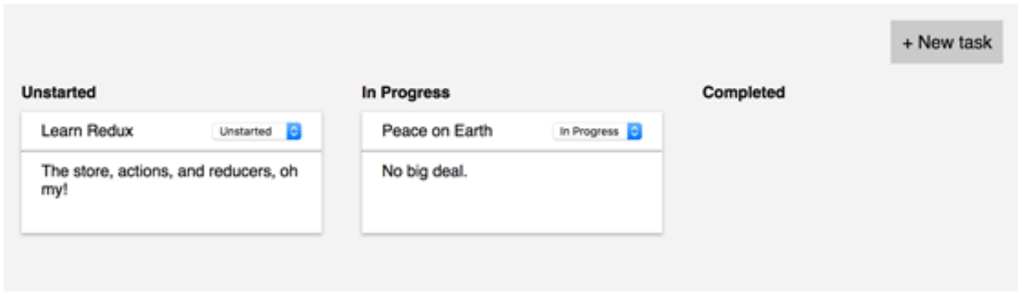


Figure 2.1 A mockup of what we'll be building in this chapter.

2.1.1 Designing the state shape

There's no single *right way* to approach problems with Redux, but we recommend taking some time to think about what the application state should look like before implementing a new feature. If React applications are just a reflection of the current state, what should our state object look like to satisfy the requirements? What properties should it have? Are arrays or objects more appropriate? These are the kinds of questions you should be asking when you approach new features. To recap, we know we need to do the following:

- Render a list of tasks
- Allow users to add items to the list
- Allow users to mark tasks as Unstarted, In Progress, or Completed

What state do we need to track to make all this possible? It turns out that store requirements are pretty straightforward: we need a list of task objects with a title, description, and status. Application state that lives in Redux is a simple JavaScript object. listing 2.1 is an example of what that object might look like.

Listing 2.1 An outline of the Redux store

```
{
  tasks: [ // #A
    {
      id: 1, // #B
      title: 'Learn Redux',
      description: 'The store, actions, and reducers, oh my!',
      status: 'In Progress',
    },
    {
      id: 2,
      title: 'Peace on Earth',
      description: 'No big deal.',
      status: 'Unstarted',
    }
  ]
}
```

#A The tasks key represents one "slice" of the data that could make up a store
#B Each task is an object with several properties

The store is simple, a `tasks` field with an array of task objects. How you organize the data in your Redux store is completely up to you, and we'll explore popular patterns and best practices later in the book.

Deciding up front how our data will look will be a big help down the road in determining what kinds of actions and reducers we might need. Remember, it may be helpful to think of client-side state like a database. Just as if we were dealing with some persistent data store like a SQL database, declaring a data model will help us organize our thoughts and drive out the code we need. Throughout the book, we'll start each new feature by revisiting this process of defining a desired state shape.

2.2 Using Create React App

React has always enjoyed a reputation for being beginner-friendly. Compared with larger frameworks like Angular and Ember, its API and feature set are small. The same cannot be said for many of the surrounding tools you'll find in many production-ready applications. This includes Webpack, Babel, ESLint, and a dozen others with varying learning curves. We developers couldn't be bothered to do all this configuration for each new project or prototype from scratch, so an abundance of starter kits and boilerplate applications were created. Although popular, many of these starter kits became wildly complex and equally intimidating for beginners to use.

Fortunately, in mid-2016, Facebook released an officially supported tool that does this complex configuration work for you and abstracts most of it away. Create React App is a command line interface (CLI) tool that will generate a relatively simple, production-ready React application. Provided you agree with enough of the choices made within the project, Create React App can easily save days of setup and configuration time. We're sold on this tool as the preferred way to get new React projects off the ground, so we'll be using it to kick-start our application.

2.2.1 Installing Create React App

Create React App is a module that can be installed via your favorite package manager. In this book, we'll be using *npm*. In a terminal window, run the following command at the prompt:

```
npm install --global create-react-app
```

Once installed, we can create a new project with:

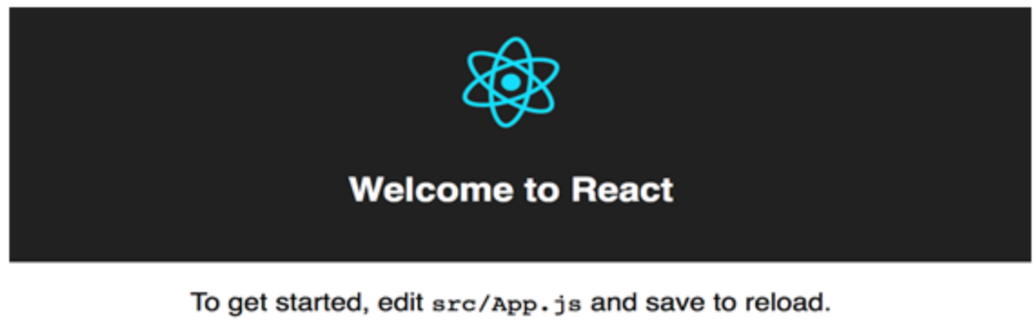
```
create-react-app parsnip
```

Creating a new application can take a few minutes, depending on the time it takes to install the dependencies on your machine. When complete, there will be a newly created `parsnip` directory waiting for you. Navigate to that directory now and we'll get up and running.

To view the application, we'll start the development server, which takes care of serving our JavaScript code to the browser (among other things). Run the following command from within the `parsnip` directory:

```
npm start
```

If `create-react-app` didn't open a browser window automatically after starting the development server, open a browser and head to localhost:3000. You should see something like figure 2.2.


Welcome to React

To get started, edit `src/App.js` and save to reload.

Figure 2.2 The home page in a browser after bootstrapping a new React application with the create-react-app command.

Go ahead and follow the instructions. Try changing the "To get started…" text by editing the `src/App.js` file. You should see the browser update automatically, without having to reload the page. Webpack is nice enough to take care of the reloading for us. We'll cover this and more development workflow enhancements in-depth in chapter 3.

2.3 Basic React Components

Before we jump into configuring Redux, let's lay some groundwork by building a few simple React components. We generally like to approach features "from the outside in," meaning we'll start by building the UI first, then hook up any necessary behavior. It helps us stay grounded in what the user will eventually experience, and the earlier you can interact with a working prototype, the better. It's much better to iron out issues with a design or feature spec early, before too much work gets underway.

We also want to make sure we're building flexible, reusable UI components. If we define our components with clear interfaces, reusing and rearranging them becomes easy. Start by creating a new directory under `src/` called `components/`, then create files for the new components, `Task.js`, `TaskList.js`, and `TasksPage.js`.

`Task` and `TaskList` will be *stateless functional components*, introduced in React v0.14. They don't have access to lifecycle methods like `componentDidMount`, only accept props, don't use `this.state` or `this.setState`, and they're defined as plain functions instead of with `createReactClass` or ES2015 classes.

These kinds of components are wonderfully simple; we don't have to worry about `this`, they're easier to work with and test, and they cut down on the number of lines of code you might need with classes. They simply accept props as input, and return some UI. What more could you ask for? Copy the code in listing 2.2 to `Task.js`.

Listing 2.2 src/components/Task.js

```
import React from 'react';

const Task = props => { // #A
  return (
    <div className="task">
      <div className="task-header">
        <div>{props.task.title}</div> // #B
      </div>
      <hr />
      <div className="task-body">{props.task.description}</div>
    </div>
  );
}

export default Task;
```

#A Stateless functional components are exported anonymous functions
#B These components receive and display props from parent components

NOTE We aren't including the contents of CSS files in this book. They're verbose and don't aid in the understanding of Redux topics. Please see the supplementary code if you would like to replicate the styles found in screenshots.

The implementation for the `TaskList` component is equally straightforward. The column name and a list of tasks will be passed in from a parent component. Copy the code in listing 2.3 to `TaskList.js`.

Listing 2.3 src/components/TaskList.js

```
import React from 'react';
import Task from './Task';

const TaskList = props => {
  return (
    <div className="task-list">
      <div className=”task-list-title”>
        <strong>{props.status}</strong>
      </div>
      {props.tasks.map(task => (
        <Task key={task.id} task={task} />
      ))}
      </div>
    );
  }
}

export default TaskList;
```

Redux allows us to implement a significant chunk of our React components as these stateless functional components. Because we get to offload most of the app's state and logic to Redux, we can avoid the component bloat that is typical of nearly all large React applications. The Redux community commonly refers to these types of components as *presentational components,* and we'll cover them more in-depth later in the chapter.

Within `TasksPage.js`, import the newly created `TaskList` component and display one for each status (listing 2.4). Although it doesn't yet, this component will need to manage local state when we introduce the new task form. For that reason, it is implemented as an ES6 class.

Listing 2.4 src/components/TasksPage.js

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

const TASK_STATUSES = ['Unstarted', 'In Progress', 'Completed']; // #A

class TasksPage extends Component { // #B
  renderTaskLists() {
    const { tasks } = this.props;
    return TASK_STATUSES.map(status => { // #C
      const statusTasks = tasks.filter(task => task.status === status);
      return <TaskList key={status} status={status} tasks={statusTasks} />;
    });
  }

  render() {
    return (
      <div className="tasks">
        <div className="task-lists">
          {this.renderTaskLists()}
        </div>
      </div>
    );
  }
}

export default TasksPage;
```

#A Tasks can have one of three states
#B ES6 classes are used when local state must be managed
#C Display one column per status, with corresponding tasks

To start, `TasksPage` will receive some mock tasks from the top-level component, `App`. `App` will also be created using an ES6 class, because it will eventually connect to the Redux store.

Listing 2.5 src/App.js

```
import React, { Component } from 'react';
import TasksPage from './components/TasksPage';

const mockTasks = [ // #A
  {
    id: 1,
    title: 'Learn Redux',
    description: 'The store, actions, and reducers, oh my!',
    status: 'In Progress',
  },
  {
    id: 2,
    title: 'Peace on Earth',
    description: 'No big deal.',
    status: 'In Progress',
  },
];

class App extends Component {
  render() {
    return (
      <div className="main-content">
        <TasksPage tasks={mockTasks} />
      </div>
    );
  }
}

export default App;
```

#A Until Redux is introduced, mock tasks will populate the UI

At this point, you should be able to run your small React application with `npm start` and view it in the browser. Bear in mind that it'll look dreadfully boring until you circle back to apply some styles. Again, you can borrow ours from the supplemental code, if you like.

2.4 Revisiting the Redux architecture

Our small React application is now ready to see Redux introduced. Before we dive straight in, lets consider the full arc of what will be required by revisiting the Redux architecture, introduced in chapter 1.

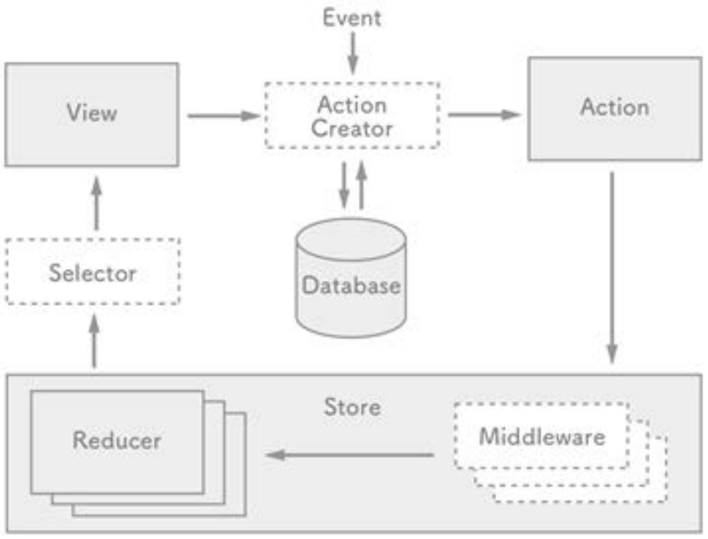


Figure 2.3 The Redux architecture.

The store is a logical starting point for introducing Redux into an application. The Redux package exposes just a few methods that facilitate the creation of a store. Once a store has been created, we'll connect it to the React application using the `react-redux` package, enabling a view (component) to dispatch actions. Actions eventually return to the store, to be read by reducers, which determine the next state of the store.

2.5 Configuring the Redux store

The main hub of functionality in Redux is the store — the object responsible for managing application state. Let's take a quick look at the store and its API in an isolated context. As an example, we'll look at a tiny program to increment a number.

2.5.1 The big picture and the store API

In reading about Redux and talking with other community members, you'll see or hear references to the store, the Redux store, or the state tree often used interchangeably. Generally, what these terms refer to is simply a JavaScript object like any other. Let's look at the API that Redux provides to interact with the store.

The Redux package exports a `createStore` function that, you guessed it, is used to create a Redux store. Specifically, the Redux store is an object with a few core methods that can be used to read and update state, and respond to any changes: `getState`, `dispatch`, and `subscribe`. We'll capture all three in the quick example in listing 2.6.

Listing 2.6 The store API in action

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

function counterReducer(state = 0, action) {  //#A
  if (action.type === 'INCREMENT') {
    return state + 1;
  }
  return state;
}

const store = createStore(counterReducer);     //#B

console.log(store.getState());  //#C

store.subscribe(() => {  //#D
  console.log('current state: ', store.getState());
});

store.dispatch({ type: 'INCREMENT' });  //#E
```

#A The store requires at least one reducer function (counterReducer)
#B Create a store with the reducer
#C Read the current state of the store
#D Do something after the store has updated
#E Send a new action to the reducers to update the store

The first argument passed to the `createStore` function is a reducer. Recall from chapter 1 that reducers are functions which inform the store how it should update state in response to actions. The store requires at least one reducer.

As promised, there are three methods on the store to show off. The first, `getState`, can be used to read the contents of the store. You will infrequently need to call this method yourself.

`subscribe` allows us to respond to changes in the store. For the sake of this example, we're logging out the newly updated state to the console. When we start connecting Redux to React, this method will be used under the hood to allow React components to re-render when any state changes in the store.

Because we can't mutate the store ourselves and only actions can result in a new state, we'll need a way to send new actions on to the reducers. `dispatch` is just that method.

2.5.2 Creating a Redux store

Back to business! In this section, we'll begin to create our store and its dependencies. Recall from the architecture diagram that a store contains one or more reducers and, optionally, middleware. We'll save middleware for a subsequent chapter, but at least one reducer is required to create a store.

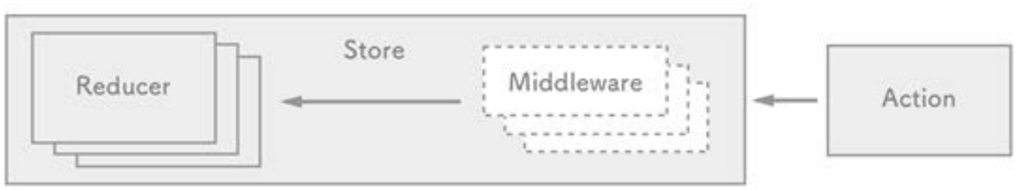


Figure 2.4 A store requires one or more reducers and may include middleware. The arrow between middleware and reducers indicates the order in which actions will eventually be handled.

Let's begin by adding Redux as a dependency of the project, then move our initial tasks data into Redux. Make sure you're in the `parsnip` directory and install the package by running the following command in a terminal window:

```
npm install -P redux
```

The `-P` flag is an alias for `--save-prod`, resulting in the package being added to our dependencies in the `package.json` file. Starting in npm5, this is the default install behavior. Now that Redux has been added, the next step is to integrate it into our existing React components. First, create the store by adding the code shown in listing 2.7 to `index.js`.

Listing 2.7 src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App';
import { createStore } from 'redux'  //#A
import tasks from './reducers' //#B
import './index.css';

const store = createStore(tasks)  //#C
…
```

#A import the `createStore` function from the redux package
#B Because we need at least one reducer to create a Redux store, import the `tasks` reducer, which we'll define in a moment
#C Create the store by passing the reducer to `createStore`

The next step is to make the store available to the React components in the app, but the code you just added isn't quite functional yet. Before going any further in `index.js`, we need to provide a barebones implementation of the `tasks` reducer.

2.5.3 The tasks reducer

As we've just learned, creating a new Redux store requires a reducer. The goal of this section is to get just enough done to create a new store, and we'll fill out the rest of the functionality as we move through the chapter.

If you recall from chapter 1, a reducer is a function that takes the current state of the store and an action, and returns the new state after applying any updates. The store is responsible for storing state, but it relies on reducers that you'll create to determine how to update that state in response to an action.

We won't handle any actions yet; we'll simply return the state without modifications. Within the `src` directory, create a new directory, `reducers`, with an `index.js` file. In this file, we're going to create and export a single function, `tasks`, that simply returns the given state.

Listing 2.8 src/reducers/index.js

```
export default function tasks(state = [], action) { //#A
  return state
}
```

#A Currently the `action` argument isn't being used, but we'll add more functionality to this reducer function once we start dispatching actions.

That's it! Do a little dance, because you've just written your first reducer. We'll be back later to make this function a bit more interesting.

2.5.4 Default reducer state

It's common to provide reducers with some initial state, which involves nothing more than providing a default value for the `state` argument in the `tasks` reducer. Before we get back to connecting the Redux store to our application, let's move the list of mock tasks out of `App.js`, and into `src/reducers/index.js`, a more appropriate place for initial state to live.

Listing 2.9 src/reducers/index.js

```
const mockTasks = [
  {
    id: 1,
    title: 'Learn Redux',
    description: 'The store, actions, and reducers, oh my!',
    status: 'In Progress',
  },
  {
    id: 2,
    title: 'Peace on Earth',
    description: 'No big deal.',
    status: 'In Progress',
  },
];

export default function tasks(state = { tasks: mockTasks }, action) {
  return state;
}
```

Don’t worry if your `App` component is breaking as a result of removing the mock data. We’ll fix that up shortly. At this point the store has the correct initial data, but we still need to somehow make this data available to the UI. Enter `react-redux`!

2.6 Connecting Redux and React with react-redux

As we discussed in chapter 1, Redux was built with React in mind, but they are two totally discrete packages. To connect Redux with React, we’ll use the React bindings from the `react-redux` package. Redux provides only the means to configure a store. `react-redux` will bridge the gap between React and Redux by providing the ability to enhance a component, allowing it to read state from the store or dispatch actions. `react-redux` gives us two primary tools for connecting our Redux store to React:

- `Provider` - a React component that you’ll render at the top of the React app. Any components rendered as children of `Provider` can be granted access to the Redux store.
- `connect` - a function used as a bridge between React components and data from the Redux store.

Pause here to install the package: `npm install -P react-redux`.

2.6.1 Adding the Provider component

`Provider` is a component that takes the store as a prop and wraps the top-level component in our app — in this case, `App`. Any child component rendered within Provider will have the ability to access the Redux store, no matter how deeply nested.

In `index.js`, import the `Provider` component and wrap the App component:

Listing 2.10 src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';  //#A
import tasks from './reducers';
import App from './App';
import './index.css';

const store = createStore(tasks);

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

#A Import the `Provider` component
#B `Provider` is now our most top-level React component. It works in conjunction with `connect` to make the store available to any child component

Think of the `Provider` component as an enabler. You won't interact with it directly very often, typically only in a file like `index.js`, which takes care of initially mounting the app to the DOM. Behind the scenes, `Provider` ensures we can use `connect` to pass data from the store to one or more React components.

2.6.2 Passing data from Redux to React components

The groundwork has been laid to pass data from the store into a React component. We have a Redux store with a `tasks` reducer, and we've used the `Provider` component from `react-redux` to make the store available to our React components. Now it's nearly time to enhance a React component with `connect`.

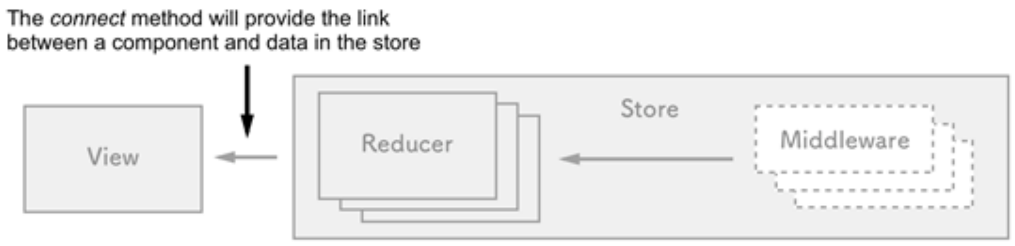

The *connect* method will provide the link between a component and data in the store
View
Reducer
Store
Middleware

Figure 2.5 The *connect* method bridges the gap between the store and views (components).

Generally, we can break visual interfaces into two major concerns, data and UI. In our case, the data is the JavaScript objects that represent tasks, and the UI is the few React components that take these objects and render them on the page. Without Redux, we'd deal with both of these concerns directly within React components.

As you can see in figure 2.6, the data we use to render our UI is moved entirely out of React and into Redux. The `App` component is will be considered an *entry point* for data from Redux. As the application grows we'll introduce more data, more UI, and as a result, more entry points. This kind of flexibility is one of Redux's greatest strengths. Our application state lives in one place, and we can pick and choose how we want that data to flow into the application.

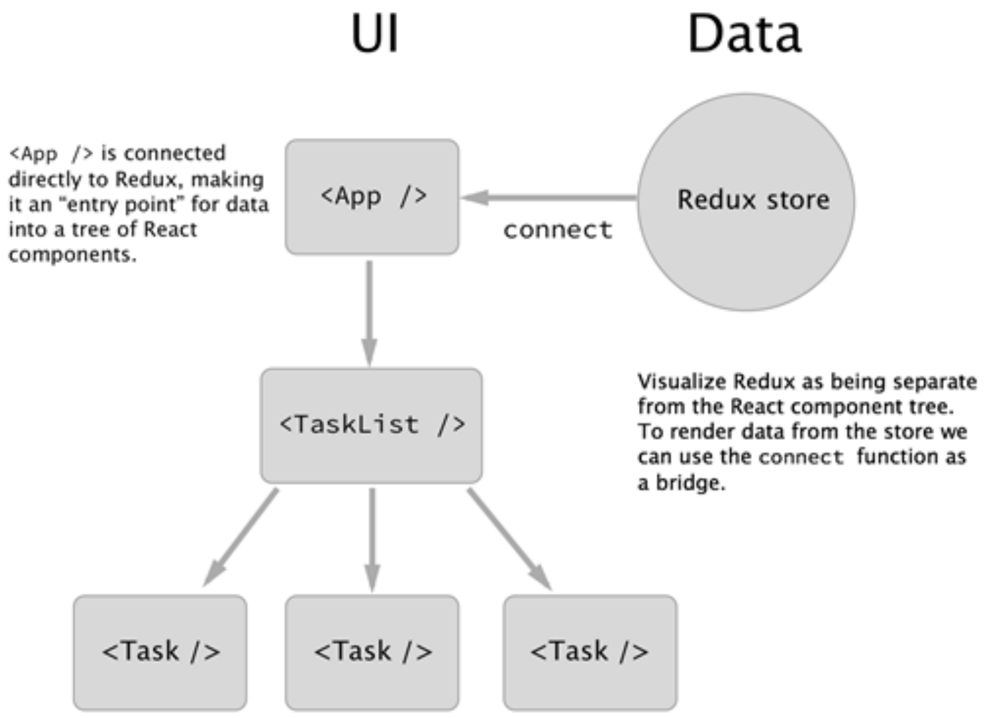

UI
Data
<App /> is connected directly to Redux, making it an "entry point" for data into a tree of React components.
<App />
connect
Redux store
<TaskList />
Visualize Redux as being separate from the React component tree. To render data from the store we can use the connect function as a bridge.
<Task />
<Task />
<Task />

Figure 2.6 A visualization of how React and Redux work together.

In listing 2.11, we're introducing a couple new concepts: `connect` and `mapStateToProps`. By adding `connect` to the App component, we've declared it as an entry point for data from the Redux store. We've only connected one component here, but as our application grows we'll start to discover best practices for when to use `connect` with additional components.

Here we're passing `connect` a single argument, the `mapStateToProps` function. Note that the name `mapStateToProps` is a convention, not a requirement. The name stuck for a reason, because it's an effective descriptor of the role of this function. State refers to the data in the store, and props are what gets passed in to the connected component. Whatever we return from `mapStateToProps` will be passed to our component as props.

Listing 2.11 src/App.js – connecting components

```
import React, { Component } from ‘react’;
import { connect } from ‘react-redux’; // #A
import TasksPage from './components/TasksPage';

class App extends Component {
  render() {
    return (
      <div className="main-content">
```

```
        <TasksPage tasks={this.props.tasks} /> // #B
      </div>
    );
  }
}

function mapStateToProps(state) { //#C
  return {
    tasks: state.tasks  //#D
  }
}

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

#A Add `connect` to the list of imports
#B Tasks will be available via `props` after connected to the store
#C The `state` argument is the entire contents of the Redux store, specifically the result of calling `getState` on the store instance
#D The return value of `mapStateToProps` will be passed into the App component as props, which is why `render` is able to reference `this.props.tasks`

Now the application is successfully rendering data from the Redux store! Notice how you didn't have to update the `TasksPage` component? That's by design. Because `TasksPage` accepts its data via props, it doesn't care what the source of those props is. They could come from Redux, from React's local state, or from another data library altogether.

2.6.3 Container and presentational components

Recall that `TaskList` is a presentational or UI component. It simply accepts data as props, and returns some output according to the markup you've defined. By using `connect` in the `App` component, we've secretly introduced their counterparts, known as *container components*.

Presentational components don't have dependencies on Redux. They don't know or care that we're using Redux to manage our application state. By using presentational components, we've introduced determinism into our view renders. In other words, given the same data, we'll always have the same rendered output. Presentational components then, are easily tested and provide our application with sweet, sweet predictability.

Presentational components are great, but something needs to know how to get data out of the Redux store and pass it to our presentational components. This is where container components, like `App`, come in. In this simple example, they have a few responsibilities:

- Get data from the Redux store via `connect`
- Use `mapStateToProps` to pass only relevant data to the component being connected
- Render presentational components

Again, separating things into container and presentational components is a convention, not a hard and fast rule that React or Redux enforces. But, it's one of the most popular and pervasive patterns for a reason. It allows us to decouple how our app *looks* from what it *does*. Defining our UI as presentational components means we have simple, flexible building blocks that are easy to reconfigure and reuse. When we're working with data from Redux, we can deal with container components without having to worry about markup. The inverse applies for when we are working with UI.

At this point, we can view the data being rendered in the browser; our app is rendering a simple list of tasks retrieved from the Redux store. Now it's time to wire up some behavior! Let's see what it takes to add a new task to the list.

2.7 Dispatching Actions

We'll follow the same workflow that we used to render the static list of tasks. We'll start with the UI, then take care of implementing functionality. Let's start with an 'New task' button and a form. When a user clicks the button, we'll render a form with two fields, title and description. Eventually, it'll look roughly like figure 2.7.

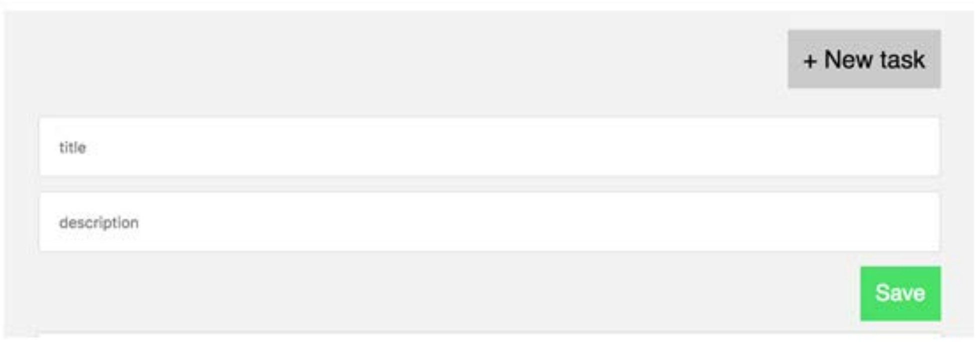


Figure 2.7 The New Task form.

Modify the code in `TasksPage.js` to match listing 2.12. This code is still plain React, so much of it may be familiar to you.

Listing 2.12 Adding the 'New Task' form

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

class TasksPage extends Component {
  constructor(props) {
    super(props);
    this.state = { //#A
      showNewCardForm: false,
      title: '',
      description: '',
    };
  }

  onTitleChange = (e) => {  //#B
    this.setState({ title: e.target.value });
  }
```

```
onDescriptionChange = (e) => {
  this.setState({ description: e.target.value });
}

resetForm() {
  this.setState({
    showNewCardForm: false,
    title: '',
    description: '',
  });
}

onCreateTask = (e) => {
  e.preventDefault();
  this.props.onCreateTask({  //#C
    title: this.state.title,
    description: this.state.description,
  });
  this.resetForm();  //#D
}

toggleForm = () => {
  this.setState({ showNewCardForm: !this.state.showNewCardForm });
}

renderTaskLists() {
  const { tasks } = this.props;
  return TASK_STATUSES.map(status => {
    const statusTasks = tasks.filter(task => task.status === status);
    return (
      <TaskList
        key={status}
        status={status}
        tasks={statusTasks}
      />;
  });
}

render() {
  return (
    <div className="task-list">
      <div className="task-list-header">
        <button
          className="button button-default"
          onClick={this.toggleForm}
        >
          + New task
        </button>
      </div>
      {this.state.showNewCardForm && (
        <form className="task-list-form" onSubmit={this.onCreateTask}>
          <input
            className="full-width-input"
            onChange={this.onTitleChange}
            value={this.state.title}
            type="text"
            placeholder="title"
          />
          <input
```

```
              className="full-width-input"
              onChange={this.onDescriptionChange}
              value={this.state.description}
              type="text"
              placeholder="description"
            />
            <button
              className="button"
              type="submit"
            >
              Save
            </button>
          </form>
        )}

        <div className="task-lists">
          {this.renderTaskLists()}
        </div>
      </div>
    );
  }
}

export default TasksPage;
```

#A It’s often simpler to use React and `setState` for UI-related state, like whether the form is open and the current values of the form inputs
#B A special syntax that ensures the value of `this` will be correct
#C Submitting the form is as simple as firing the onCreateTask prop
#D We’ll also make sure to reset the form’s state after submission

Our `TaskList` component is now keeping track of some local state—whether the form is visible, and the text values in the form. The form inputs are what’s known in React as *controlled components*. All that means is the values of the input fields are set to the corresponding local state values, and for each character typed into the input field, local state is updated. When a user submits the form to create a new task, we’ll call the `onCreateTask` prop to indicate an event has taken place. Because we’re calling `onCreateTask` from `this.props`, we know that this function needs to be passed down from the parent component, `App`.

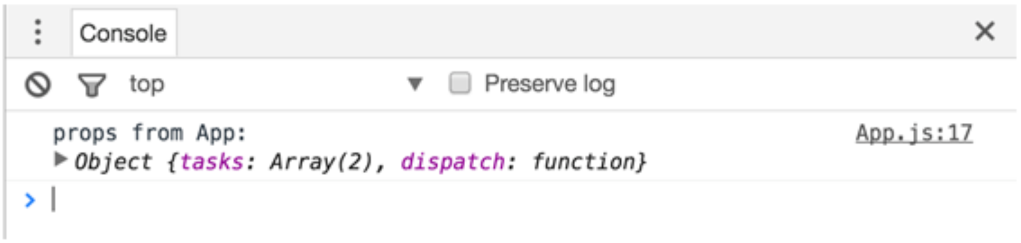
POP QUIZ **What’s the only way to initiate a change in the Redux store? (No peeking at this section title.) Dispatching an action is exactly right. We have a good idea, then, of how we should implement the `onCreateTask` function: we need to dispatch an action to add a new task.**

In `App.js`, we know that `App` is a connected component and is enhanced with the ability to interact with the Redux store. Do you remember which of the store APIs can be used to send off a new action? Take a moment and log the value of `this.props` in the render method of App, as shown in listing 2.13.

Listing 2.13 Logging this.props in src/App.js

```
…
render() {
   console.log('props from App: ', this.props) // #A
   return (
     ...
   )
  }
}
…
```

#A Log the value of `this.props` at the top of the `render` method



```
props from App:
▶Object {tasks: Array(2), dispatch: function}
```

Figure 2.8 Console output from logging the props available to the App component.

There it is: a `dispatch` prop in addition to our expected `tasks` array. What is `dispatch`? We know that the store is extremely protective of its data. It only provides us one way to update state — dispatching an action. `dispatch` is part of the store's API, and `connect` conveniently provides this function to our component as a prop. Let's create a handler where we dispatch a `CREATE_TASK` action. The action will have two properties:

- `type` — a string that represents the category of action being performed. By convention, they are capitalized and use underscores as delimiters. This is the only required property for an action to be considered valid.
- `payload` — an object that provides the data necessary to perform the action. Having a `payload` field is optional and can be omitted if no additional data is required to perform the action. For example, an action to log a user out may just contain a type of `LOGOUT`, with no additional data requirements. If additional data is required however, any keys may be passed in the action. The name `payload` isn't required by Redux, but is a popular organizational convention that we'll stick with throughout the book. The pattern is commonly referred to as Flux Standard Actions (FSA) and more details can be found in this GitHub repository: https://github.com/acdlite/flux-standard-action.

Listing 2.14 src/App.js – adding an action handler

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import TasksPage from './components/TasksPage';

class App extends Component {
  onCreateTask = ({ title, description }) => {
    this.props.dispatch({  //#A
      type: 'CREATE_TASK',
      payload: {
        title,
        description
      }
    });
  }

  render() {
    return (
      <div className="main-content">
        <TasksPage
          tasks={this.props.tasks}
          onCreateTask={this.onCreateTask} // #B
        />
      </div>
    );
  }
}
```

#A `this.props.dispatch`, injected by `connect`, allows us to dispatch an action to the store
#B The `onCreateTask` handler is passed to `TasksPage` as a simple callback prop

This listing also illustrates one of the other main roles of container components: action handling. We don't want `TasksPage` to worry about the details of creating a new task, it need to only indicate that the user wishes to do so by firing the `onCreateTask` prop.

2.8 Action Creators

We dispatched the `CREATE_TASK` action object directly in the previous example, but it's not something we usually do outside of simple examples. Instead, we'll invoke *action creators* — functions that return actions.

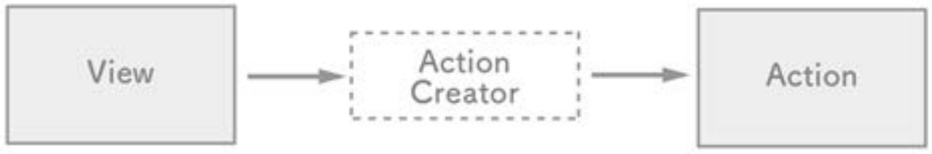


Figure 2.9 Although views can dispatch actions, they will often invoke action creators instead — functions that return actions.

Actions and action creators are closely related and work together to dispatch actions to the store, but they fulfill different roles:

- Actions: objects that describe an event
- Action creators: functions that return actions

Why use action creators? Action creators have a friendlier interface; all you need to know is which arguments the action creator function expects. You won't have to worry about specifics, like the shape of the action's payload or any logic that might need to be applied before the action can be dispatched. By the same token, an action creator's arguments are helpful because they clearly document an action's data requirements.

Later in the book, we'll be implementing a good chunk of our application's core logic directly within action creators. They'll do things like make AJAX requests, perform redirects, and create in-app notifications.

2.8.1 Using action creators

From the last section, we know `dispatch` accepts an action object as an argument. Instead of dispatching the action directly, we'll use an action creator. Within the `src` directory, create a new directory called `actions`, with an `index.js` file within it. This file is where our action creators and actions will live. Add the code in listing 2.15 to that newly created file.

Listing 2.15 src/actions/index.js — the createTask action creator

```
let _id = 1;
export function uniqueId() {  //#A
  return _id++;
}

export function createTask({ title, description }) {  //#B
  return {
    type: 'CREATE_TASK',
    payload: {  //#C
      id: uniqueId(),
      title,
      description,
      status: 'Unstarted',
    }
  };
}
```

#A `uniqueId` is a utility function to generate numeric ids for tasks. When we hook up the app to a real server in Chapter 4 this will no longer be necessary
#B The function signature makes it clear that a title and a description are required to dispatch the CREATE_TASK action
#C The `payload` property contains all the data necessary to perform the action

There's one piece of cleanup you'll need to do after adding the `uniqueId` function. Update `src/reducers/index.js` to use `uniqueId` instead of hard-coded IDs. This will ensure your task ids will increment correctly as you create them, and we'll use these ids when we allow users to edit tasks later in the chapter.

Listing 2.16 src/reducers/index.js

```
import { uniqueId } from '../actions'; //#A

const mockTasks = [
  {
    id: uniqueId(), //#B
    title: 'Learn Redux',
    description: 'The store, actions, and reducers, oh my!',
    status: 'In Progress',
  },
  {
    id: uniqueId(),
    title: 'Peace on Earth',
    description: 'No big deal.',
    status: 'In Progress',
  },
];
```

#A Import the `uniqueId` function you just created in `src/actions/index.js`
#B Use `uniqueId` instead of hard-coded ids

To finish the implementation, update the code in App.js to import and use our new action creator.

Listing 2.17 src/App.js

```
...
import { createTask } from './actions'; // #A

class App extends Component {
 ...
 onCreateTask = ({ title, description }) => {
   this.props.dispatch(createTask({ title, description })); // #B
 }
 ...
}
...
```

#A Import the action creator
#B Instead of passing an action object to `this.props.dispatch`, we'll pass the action creator

To recap, the `App` container component has access to the `dispatch` method, thanks to `connect`. `App` imports an action creator, `createTask`, and passes it a `title` and a `description`. The action creator formats and returns an action. In the next section, we'll follow that action through to the reducer and beyond.

Remember that `uniqueId` function? How we generate the `id` field is particularly noteworthy, because it introduces a *side-effect*.

DEFINITION A side-effect is any code that has a noticeable effect on the outside world, like writing to disk or mutating some data. Put another way, it's code that does anything but take inputs and return a result.

Functions with side-effects do something other than just return a value. `createTask` mutates some external state — the ID that we increment whenever we create a new task.

2.8.2 Action creators and side-effects

Most of the code we've written thus far has been *deterministic*, meaning it produces no side-effects. This is all well and good, but we need to deal with side-effects somewhere. Code that simply operates on data is easy to work with and think about, but side-effects are necessary to do anything useful. Eventually we'll need to do things like write to the browser's local storage and communicate with a web server. Both are considered side-effects and are totally ubiquitous in the world of web applications.

We know we can't do a whole lot without side-effects. What we *can* do is isolate them by enforcing good practices around where they can be performed. Reducers must be pure functions, so they're out. So, you guessed it, that leaves action creators! `createTask` is non-deterministic, and that's perfectly okay. Chapters 4, 5, and 6 will explore various strategies for managing side-effects.

2.9 Handling actions with reducers

We defined a simple tasks reducer when we used `createStore` to initialize our Redux store, but at this point it simply returns the current state.

Listing 2.18 src/reducers/index.js

```
...
export default function tasks(state = { tasks: mockTasks }, action) {
  return state;
}
```

This reducer is completely valid and functional, but doesn't do anything particularly useful. The real point of reducers is to handle actions. *Reducers* are functions that accept the store's current state and an action, and return the next state after applying any relevant updates. We're still missing that last bit: we need to actually change our state.

The store's role is to manage application state; it's where the data lives, it controls access, and it allows components to listen for updates. What it doesn't, and can't do, is define how exactly its state should change in response to actions. That is up to us to define and reducer*s* are the mechanism Redux gives us to accomplish this.

2.9.1 Responding to actions in reducers

We're correctly dispatching the `CREATE_TASK` action, indicating an event has occurred. But the action doesn't specify how to handle this event. How should state update in response to the action? We chose to store our task objects in an array, so all we need to do is push an element on to the list. Reducers check the action's type to determine if it should respond to it. This amounts to a simple conditional statement that describes how the state should update for a given action type.

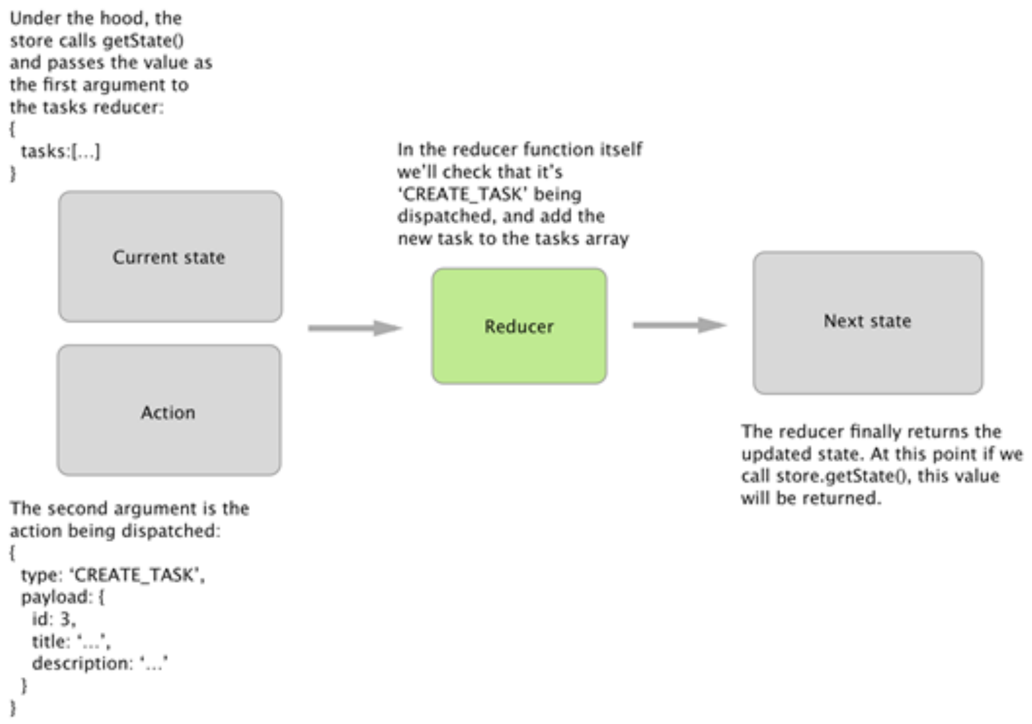

Under the hood, the store calls getState() and passes the value as the first argument to the tasks reducer:
{
tasks:[...]
}
Current state
Action
The second argument is the action being dispatched:
{
type: 'CREATE_TASK',
payload: {
id: 3,
title: '...',
description: '...'
}
}
In the reducer function itself we'll check that it's 'CREATE_TASK' being dispatched, and add the new task to the tasks array
Reducer
Next state
The reducer finally returns the updated state. At this point if we call store.getState(), this value will be returned.

Figure 2.10 A look at our reducer in action. It takes two arguments, the current state of the store and the CREATE_TASK action, and returns the next state.

In our case, if the reducer receives an action of type `CREATE_TASK`, we expect the next state tree to have one more task in the list, but be otherwise identical to the previous state. An action of any other type will result in an unchanged Redux store, because `CREATE_TASK` is all we're listening for so far.

Update the `tasks` reducer to handle the `CREATE_TASK` action.

Listing 2.19 src/reducers/index.js

```
...
export function tasks(state = { tasks: mockTasks }, action) {
  if (action.type === 'CREATE_TASK') {  //#A
    return { tasks: state.tasks.concat(action.payload) };  //#B
  }

  return state;  //#C
}
```

#A Check if the action type is one that we care about
#B If the action is `CREATE_TASK`, add the task to the array and return the result

#C Always fall back to returning the given state in case a reducer receives an action it doesn't care to handle

Now the tasks reducer actually updates state in response to an action. As we continue to add functionality and dispatch new actions, we'll add more code like this which checks for a specific action type and conditionally applies any updates to application state.

At this point, we've completed an entire cycle within Redux's unidirectional data flow! Once the store updates, our connected component, `App`, becomes aware of the new state and performs a new render. Let's review the architecture diagram one last time to help it all sink in:

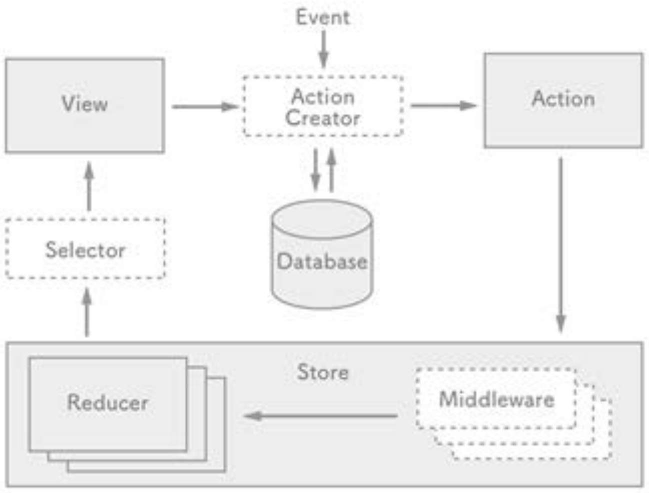


Figure 2.11 The Redux architecture

We started by creating a store, passing in the tasks reducer as an argument. After being connected to the store, the views rendered the default state specified by the tasks reducer. When a user wants to create a new task, the connected component dispatches an action creator. That action creator returns an action containing a `CREATE_TASK` type and some additional data. Finally, the reducer listens for the `CREATE_TASK` action type and determines what the next application state should look like.

2.10 Exercise

More than anything, we want to help you develop the intuition that will help you solve unique problems on your own. You now know about the store, actions, reducers, and what roles they play. Using what you've learned from implementing task creation, try making Parsnip even more awesome by allowing users to update the status of each task.

Tasks have a `status` field, which can be one of three values: *Unstarted*, *In Progress*, and *Completed*. If you open the browser to localhost:3000 you'll see the UI already displays the status of each task, but users should now be able to open a dropdown and a choose a new status.

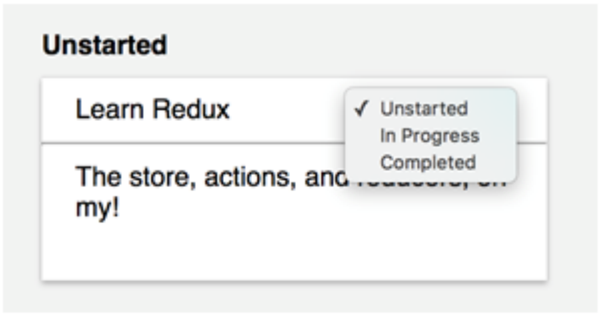


Figure 2.12 An example status dropdown.

Try your hand first at an implementation, then we'll walk through how we might approach it. If the task seems daunting, try breaking the problem down until you have manageable, actionable steps. Before getting into any code, keep a few questions in mind:

- What am I allowing users to *do*? What UI do I need to build to allow them to access these features?
- Based on my requirements, what state might I need to track to fulfill them?
- When and how does that state change?

2.11 Solution

As always, we'll start with a high-level description of what we want to accomplish, then work piece-by-piece towards an implementation. Our goal is to allow users to update a task's *status* by selecting either *Unstarted, In Progress,* or, *Completed* from a select input. Let's break down the problem into manageable chunks:

- Add a select input with the three available statuses. Tasks already have a `status` field and we can declare the possible states as a constant.
- When the user chooses a new status, dispatch an `EDIT_TASK` action with two pieces of data: the `id` of the task being updated and the desired `status`.
- The `tasks` reducer should handle `EDIT_TASK`, update the `status` of the correct task, and return the updated state tree.
- The view should re-render with the newly updated status.

Have you noticed how we tend to implement features in a particular order? It lines up nicely with the idea of a unidirectional data flow, one of the fundamental ideas in React and

Redux. A user interaction triggers an action, we handle the action, and close the loop by re-rendering the view with any updated state.

2.11.1 The status dropdown

Start by adding the status dropdown to the `Task` component.

Listing 2.20 src/components/Task.js

```
import React from 'react'

const TASK_STATUSES = [  //#A
  'Unstarted',
  'In Progress',
  'Completed'
]

const Task = props => {
  return (
    <div className="task">
      <div className="task-header">
        <div>{props.task.title}</div>
        <select value={props.task.status}> //#B
          {TASK_STATUSES.map(status => (
            <option key={status} value={status}>{status}</option>
          ))}
        </select>
      </div>
      <hr />
      <div className="task-body">{props.task.description}</div>
    </div>
  )
}

export default Task;
```

#A Define the list of possible statuses as a variable for clarity convenience
#B Add the status dropdown using the `select` and `option` elements

Now the user can interact with a dropdown that renders the correct values, but the task won’t be updated when an option is selected.

> **TIP** For the sake of simplicity, we defined `TASK_STATUSES` directly in the `Task` component, but it’s a common convention to define constants like these in a separate file.

2.11.2 Dispatching an edit action

To indicate an event has occurred in our application — the user selecting a new status for a task — we’ll dispatch an action. Create and export an action creator that builds the `EDIT_TASK` action. This is where we’ll determine the arguments to the action creator (`editTask`), and the shape of the action payload.

Listing 2.21 src/actions/index.js

```
...
export function editTask(id, params = {}) {  //#A
  return {
    type: 'EDIT_TASK',
    payload: {
      id,
      params
    }
  };
}
```

#A By using an action creator we're able to clearly communicate that the `EDIT_TASK` requires two arguments: the `id` of which task to edit, and a `params` object with any fields being updated.

Next, import `editTask` in `App`, our container component, add any necessary action handling, and pass down an `onStatusChange` prop to eventually be fired by the `Task` component.

Listing 2.22 src/App.js

```
...
import { createTask, editTask } from './actions';  // #A

class App extends Component {
 ...
 onStatusChange = (id, status) => {
   this.props.dispatch(editTask(id, { status }));  //#B
 }

 render() {
   return (
     <div className="main-content">
       <TasksPage
         tasks={this.props.tasks}
         onCreateTask={this.onCreateTask}
         onStatusChange={this.onStatusChange}  //#C
       />
     </div>
   );
 }
}
…
```

#A Import the new action creator
#B Create the `onStatusChange` handler which dispatches the `editTask` action creator
#C Pass `onStatusChange` down to `TaskList`

Next, move on to the `TasksPage` component and pass `onStatusChange` down to `TaskList` and finally on to `Task`.

Listing 2.23 src/components/TasksPage.js

```
...
return (
  <TaskList
    key={status}
    status={status}
    tasks={statusTasks}
    onStatusChange={this.props.onStatusChange}  // #A
  />
);
...
```

#A Task will ultimately be what calls `this.props.onStatusChange` with the correct arguments, so `TaskList` only needs to forward this prop along

To reach the Task component, onStatusChange will need to travel through one more component: TaskList.

Listing 2.24 src/components/TaskList.js

```
...
{props.tasks.map(task => {
  return (
    <Task
      key={task.id}
      task={task}
      onStatusChange={props.onStatusChange} // #A
    />
  );
}
...
```

#A onStatusChange needs to be passed once more as a prop to reach Task

Finally, in the Task component we can fire the `props.onStatusChange` callback when the value of the status dropdown changes.

Listing 2.25 src/components/Task.js

```
...
const Task = props => {
 return (
   <div className="task">
     <div className="task-header">
       <div>{props.task.title}</div>
       <select value={props.task.status} onChange={onStatusChange}>  // #A
         {TASK_STATUSES.map(status => (
           <option key={status} value={status}>{status}</option>
         ))}
       </select>
     </div>
     <hr />
     <div className="task-body">{props.task.description}</div>
   </div>
 );

 function onStatusChange(e) {  // #B
   props.onStatusChange(props.task.id, e.target.value)
 }
}
...
```

#A Add a callback to be run when the dropdown's change event fires
#B Call `onStatusChange` with the id of the task being updated, and the value of the new status

The only thing missing at this point is update logic. An action is being dispatched which describes an intent to edit a task, but the task itself still needs to be updated by a reducer.

2.11.3 Handling the action in a reducer

The last step is to specify how the task should be updated in response to the `EDIT_TASK` action being dispatched. Update the `tasks` reducer to check for the newly created `EDIT_TASK` action, and update the correct task.

Listing 2.26 src/reducers/index.js

```
...
export function tasks(state = initialState, action) {
  ...
  if (action.type === 'EDIT_TASK') {  // #A
    const { payload } = action;
    return {
      tasks: state.tasks.map(task => {  // #B
        if (task.id === payload.id) {
          return Object.assign({}, task, payload.params);  // #C
        }

        return task;
      })
    }
  }
}
```

#A Check if the action being passed in has a `type` that we want to handle
#B Because the list of tasks is stored as an array, to update the right task iterate over the list of tasks with map, and if the current task matches the id from the payload, update it with the new params
#C Use `Object.assign` to update the task object by returning a new copy, not modifying the original object

First, we check if the action being passed in is of type `EDIT_TASK`. If so, we iterate over the list of tasks, updating the relevant task and returning the remaining tasks without modification.

That completes the feature! Once the store updates, the connected components will perform another render and the cycle is ready to begin again.

2.12 Summary

We've implemented a couple relatively straightforward features, but in the process, we saw most of the core elements of Redux in action. It can feel like a lot, but it's not critical (or even feasible) that you leave chapter 2 with an ironclad understanding of every new concept we've introduced. We'll cover many of these individual ideas and techniques in great depth later in the book. Let's take a quick look back at what we've learned so far:

- Container components receive data from Redux and dispatch action creators, and presentational components accept data as props and handle markup.
- Actions are objects describing an event. Action creators are functions that return actions.
- Reducers are pure functions that update state in response to actions.
- Side-effects can be handled in action creators. Reducers however, should be pure functions, meaning they don't perform any mutations and always return the same value given the same inputs.
- A configured Redux store can be made available to our app via react-redux and the Provider component.
- `connect` and `mapStateToProps` are used to pass data from Redux into a React component as props.

Next up is a deep dive into the tools that make Redux developers such happy campers.

3 Debugging Redux applications

This chapter covers

- **Working with the Redux developer tools**
- **Understanding the role of monitors**
- **Using hot module replacement**

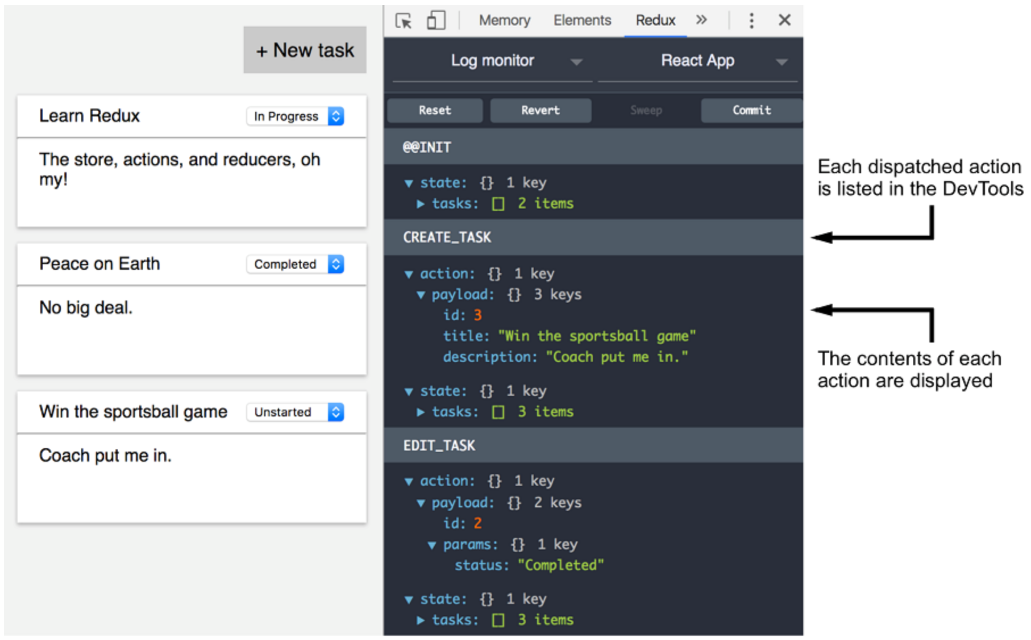
Debugging isn't just a thing you do when you're given a bug report. The same tools and developer experience are essential to developing new features, too. In chapter 1, we learned that Redux was born out of a desire for a better client-side development experience. In this chapter, we'll talk about a few areas where the Redux developer tools can provide better insight into an application, save valuable developer hours, and make for a more enjoyable day on the job.

Historically, time spent tracking down unexpected behavior could be one of the most egregious time sinks in a developer's day. Chasing state changes in a complex application using two-way databinding has sunk many developer days — we should know. However, the Flux architecture pattern successfully reduces some of the mental overhead required to keep up with state changes, thanks to its unidirectional dataflow. Standardizing on actions as the vehicles of state change introduced a certain clarity: regardless of what initiated it, a change in state can be traced back to an action.

As you've learned in the first two chapters, the list of actions dispatched in an application forms what can be thought of as a transaction log. When viewed sequentially, they can tell the story of a user's interaction with the application fairly comprehensively. Wouldn't it be nice to visualize those actions in real time? To see them as they're dispatched and the data they contain?

3.1 Introducing the Redux DevTools

The Redux developer tools, or DevTools for short, augment your development environment for real-time visualization of actions. Let's take a look at what the developer tools might look like in the context of Parsnip, the task management application we started in chapter 2.


Memory
Elements
Redux
Log monitor
React App
Reset
Revert
Sweep
Commit
+ New task
Each dispatched action is listed in the DevTools
The contents of each action are displayed

Learn Redux — In Progress
The store, actions, and reducers, oh my!

Peace on Earth — Completed
No big deal.

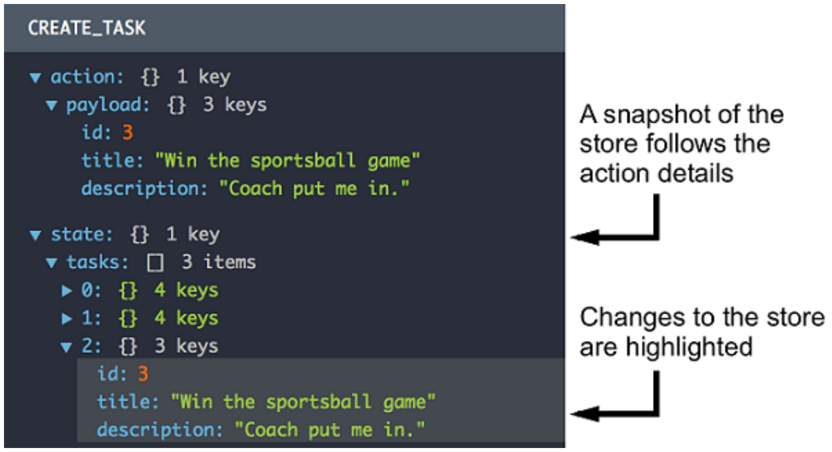
Win the sportsball game — Unstarted
Coach put me in.

```
@@INIT
▾ state: {} 1 key
  ▸ tasks: [] 2 items
CREATE_TASK
▾ action: {} 1 key
  ▾ payload: {} 3 keys
      id: 3
      title: "Win the sportsball game"
      description: "Coach put me in."
▾ state: {} 1 key
  ▸ tasks: [] 3 items
EDIT_TASK
▾ action: {} 1 key
  ▾ payload: {} 2 keys
      id: 2
    ▾ params: {} 1 key
        status: "Completed"
▾ state: {} 1 key
  ▸ tasks: [] 3 items
```

Figure 3.1 The Redux DevTools can be used to display actions in real time.

In the right page (figure 3.1), you can see a highlighted list of items — the actions that have been dispatched to produce the state displayed. Given this list of actions, you can tell exactly what we've been up to within Parsnip without having to observe us do it: the app was initialized, a third task was created, and then the second task was edited. You can see why it might be handy to have immediate feedback on each new action as they're dispatched. For every action, you can be sure that the payload looks just the way you intended. Still, there's much more that you can do with the DevTools.

You'll notice that below each action in the right page is a snapshot of the Redux store. Not only can we view the action produced, we can see the application state that resulted from that action. Better still, we can dig into the Redux store and see, highlighted, the exact values that changed as a result of the action.



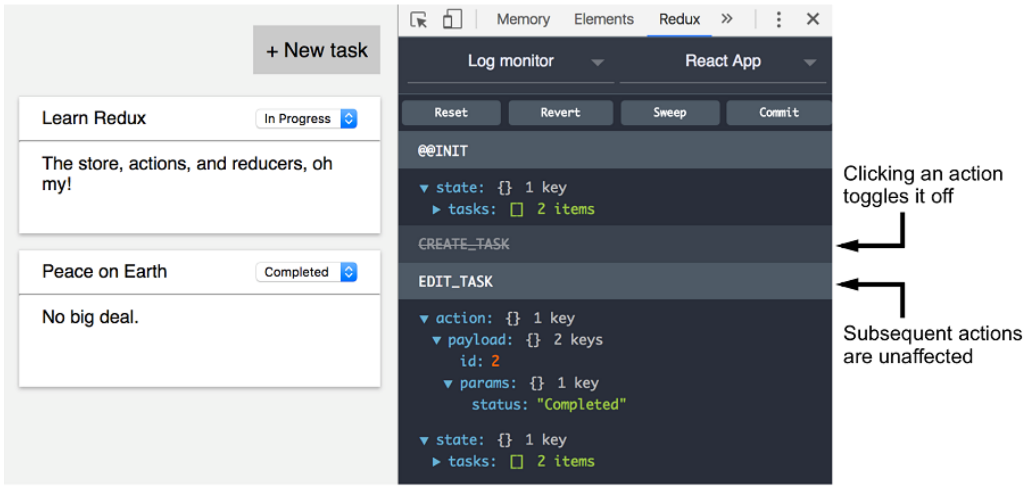
```
CREATE_TASK
▼ action: {} 1 key
  ▼ payload: {} 3 keys
      id: 3
      title: "Win the sportsball game"
      description: "Coach put me in."
▼ state: {} 1 key
  ▼ tasks: [] 3 items
    ▶ 0: {} 4 keys
    ▶ 1: {} 4 keys
    ▼ 2: {} 3 keys
      id: 3
      title: "Win the sportsball game"
      description: "Coach put me in."
```

A snapshot of the store follows the action details
Changes to the store are highlighted

Figure 3.2 The DevTools highlight attributes of the Redux store that have changed as a result of the action.

3.2 Time-travel debugging

But wait, there’s more! Clicking the title of an action in the DevTools toggles that action off. The application state is recalculated as if that action was never dispatched, even if the action is in the middle of long list of actions. Click it again, and the application returns to its original state.



Learn Redux — In Progress
The store, actions, and reducers, oh my!

Peace on Earth — Completed
No big deal.

```
@@INIT
▼ state: {} 1 key
  ▶ tasks: [] 2 items
~~CREATE_TASK~~
EDIT_TASK
▼ action: {} 1 key
  ▼ payload: {} 2 keys
      id: 2
    ▼ params: {} 1 key
        status: "Completed"
▼ state: {} 1 key
  ▶ tasks: [] 2 items
```

Figure 3.3 Toggling an action recalculates the application state as if the action was never dispatched.

The rewinding and replaying of actions like this is what inspired the name *time-travel debugging*. To determine what Parsnip would look like if we had not created a third task, there's no need to refresh and re-create the state — just hop back in time by disabling that action.

One extension for the DevTools even provides a slider UI to scroll back and forward through actions, seeing their immediate effect on the application. Note that no extra configuration or dependency is required to use time-travel debugging; it is a feature of the DevTools. More specifically, it is a feature of many DevTool monitors.

3.3 Visualizing changes with DevTools monitors

The Redux DevTools provide an interface to the actions and Redux store but don't actually provide a way to visualize that data. That work is left for monitors. This is a conscious decision to separate the two concerns, enabling the community to plug and play their own visualizations of the data to best fit their needs. Figure 3.4 illustrates this idea conceptually; one or more monitors can be configured to display the data provided by the DevTools.

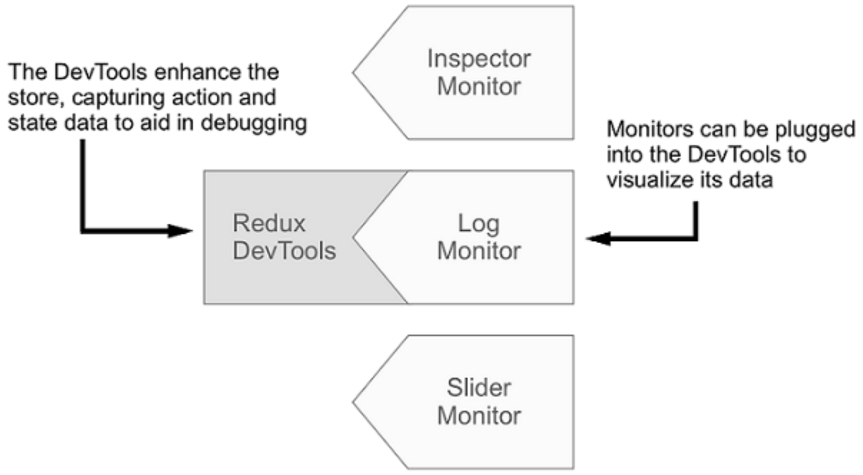

The DevTools enhance the store, capturing action and state data to aid in debugging
Inspector Monitor
Monitors can be plugged into the DevTools to visualize its data
Redux DevTools
Log Monitor
Slider Monitor

Figure 3.4 Various monitors can be combined with the Redux DevTools to visualize actions and store data.

Several of these monitors, including those listed in Figure 3.4, are open source libraries and ready for use. In the screenshots from the previous sections, you've been viewing the so-called default monitor, the *Log Monitor*. Other monitors worth mentioning are the *Slider Monitor*, described in the previous section, and the *Inspector Monitor*, the default monitor of the Redux DevTools Chrome extension. Inspector provides a user interface similar to the Log Monitor, but allows for easier filtering of actions and store data.

NOTE Several more monitors can be found in the README of the Redux DevTools repository on GitHub (https://github.com/gaearon/redux-devtools). Again, the DevTools will feed the data to any monitor, so if you can't find a monitor feature you're looking for, that may be your cue to build your own. The open source community will thank you for your contribution!

3.4 Implementing the Redux DevTools

Let's say that you've just been tasked with implementing the DevTools in your budding new application, Parsnip. The first choice you have to make is how you'd like to view the monitors. There are a few popular options:

- In a component within the application
- In a separate popup window
- Inside your browser's developer tools

For a few reasons, our personal preference is the last option — to use the Redux DevTools via the Chrome browser developer tools. First, the installation is easier than any other option. Second, the integration with our existing development workflow is seamless. Finally, the extension includes a robust set of monitors that continues to meet our needs out of the box.

As JavaScript developers, many of us already spend a lot of time within the Chrome DevTools — inspecting elements, using breakpoints, flipping between panels to check the performance of requests, and so on. Installing the Redux DevTools Chrome plugin just adds one more panel, and clicking it reveals the Inspector and other monitors. We don't miss a beat.

NOTE Redux and Chrome both refer to their developer tools by the abbreviation "DevTools." References to the Redux DevTools within the Chrome DevTools can get confusing, so pay extra attention to the difference. Going forward, we'll specify which we're referring to.

There are two steps to this process: installing the Chrome browser extension and hooking the Redux DevTools into the store. Installing the Chrome extension is the simpler of the two. Just visit the Chrome Web Store, search for Redux DevTools, and install the first package by the author remotedev.io.

On to the second step, adding the Redux DevTools to the store. Although this configuration can be done without another dependency, the `redux-devtools-extension` package is a friendly abstraction that reads like English. We'll download and instrument the package now. Install and save the package to your development dependencies with the following command:

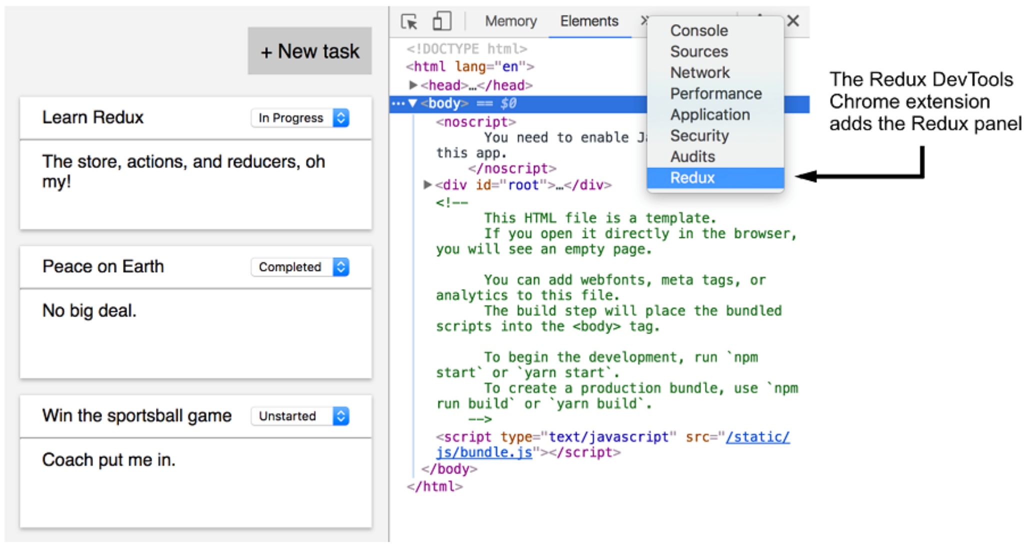
```
npm install –D redux-devtools-extension
```

Once installed, we'll import and pass a function called `devToolsEnhancer` to the store. As a refresher, Redux's `createStore` function takes up to three arguments: a reducer, an initial state, and an enhancer. In the case that only two arguments are passed, it is presumed the second argument is an enhancer and there is no initial state. Listing 3.1 is an example of this case. Enhancers are a way to augment the Redux store and the `devToolsEnhancer` function is doing just that: connecting the store with the Chrome extension to provide additional debugging features.

Listing 3.1 src/index.js

```
import { devToolsEnhancer } from 'redux-devtools-extension';
…
const store = createStore(tasks, devToolsEnhancer());
…
```

After you've completed adding the Redux DevTools enhancer to the `createStore` method, you can begin to use the tools. Flip back to the app in the browser and open the Chrome developer tools. If you're unfamiliar, from the Chrome navigation bar, select *View*, then *Developer*, and finally *Developer Tools*. The Chrome DevTools will open in a separate pane in your browser, typically with the *Elements* panel displayed by default. From the navigation bar within the Chrome developer tools, the Redux DevTools can be found by selecting the new *Redux* panel, made available by the Redux DevTools Chrome extension.

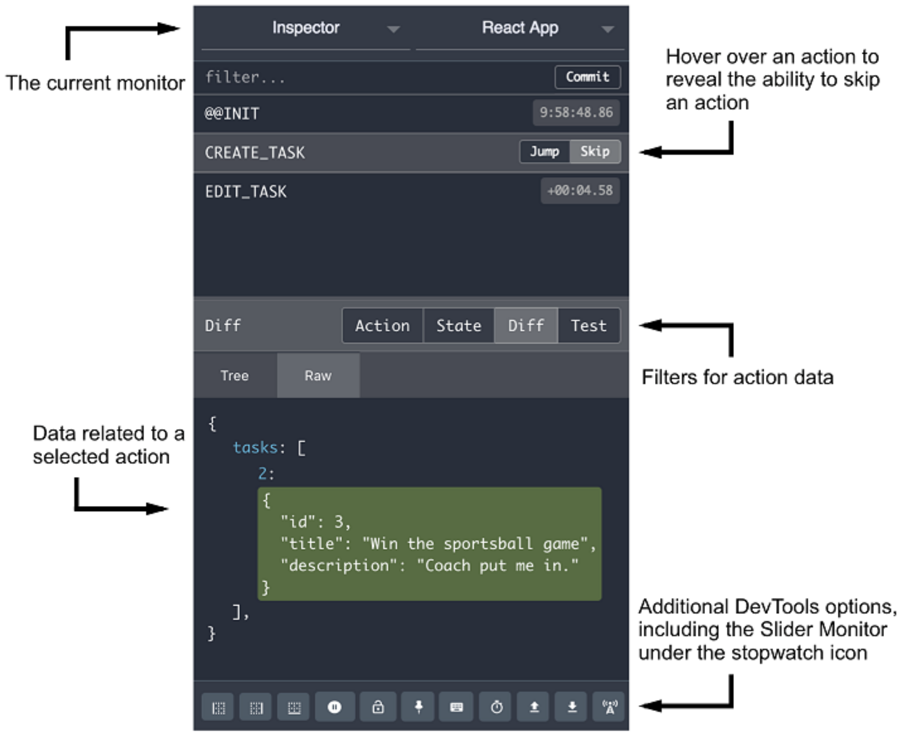


```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body> == $0
    <noscript>
        You need to enable J
this app.
    </noscript>
  ▶<div id="root">…</div>
    <!--
        This HTML file is a template.
        If you open it directly in the browser,
    you will see an empty page.

        You can add webfonts, meta tags, or
    analytics to this file.
        The build step will place the bundled
    scripts into the <body> tag.

        To begin the development, run `npm
    start` or `yarn start`.
        To create a production bundle, use `npm
    run build` or `yarn build`.
    -->
    <script type="text/javascript" src="/static/
    js/bundle.js"></script>
  </body>
</html>
```

Figure 3.5 Navigate to the Redux DevTools via the Redux panel in the Chrome developer tools.

Once you've navigated to the Redux DevTools, you should see the Inspector Monitor by default, which you can confirm by verifying that the upper left menu of the tools reads *Inspector*. If you've followed along and implemented the Redux DevTools in Parsnip, test them out by adding a couple new tasks. You should see a new action listed in the Inspector Monitor for each task added. Click the skip button for one of the actions in the monitor to toggle that action off and notice the removal of that task in the user interface. Click the action's skip button once more to toggle it back on.

When an action is selected, the other half of the Inspector Monitor will display some data about the action or the Redux store, depending on the filter you've selected. The *Diff* filter is particularly helpful for visualizing the impact that an action had on the store. The menu in the upper left will change the display between the Inspector, Log, and Chart monitors. A button, near the bottom on the panel, with a stopwatch icon opens a fourth monitor: the Slider Monitor. Pause here to take some time to explore these tools pointed out in figure 3.6, because they make for a delightful developer experience and will save your backside more times than you'll be able to count.



Inspector
React App
The current monitor
filter...
Commit
@@INIT
9:58:48.86
CREATE_TASK
Jump
Skip
EDIT_TASK
+00:04.58
Hover over an action to reveal the ability to skip an action
Diff
Action
State
Diff
Test
Filters for action data
Tree
Raw
Data related to a selected action
Additional DevTools options, including the Slider Monitor under the stopwatch icon

```
{
  tasks: [
    2:
    {
      "id": 3,
      "title": "Win the sportsball game",
      "description": "Coach put me in."
    }
  ],
}
```

Figure 3.6 The skip, diff filter, and monitor menu options offer unique ways to visualize and debug state effects.

If you think you'd prefer to use the Redux DevTools in a component within your app, the setup process is slightly more long-winded. You'll likely make use of the Dock Monitor—a component that can be shown or hidden in your application and that displays the Log Monitor, or another monitor, within it. Full instructions can be found in the README of the Redux DevTools repository on GitHub at https://github.com/gaearon/redux-devtools.

3.5 The role of Webpack

Are you already bored with your new debugging superpowers and looking for something else to optimize? As a JavaScript developer, you may be all too familiar with this workflow:

1. Place a *debugger* statement in an uncertain area of the code.
2. Click through the application until the *debugger* is triggered.
3. In the console, figure out what code should be written to make incremental progress.
4. Add the new code to your application and delete the *debugger*.
5. Return to step 1 and repeat until the bug fix or feature is complete.

Though *debugger* can be wildly more efficient than just using console logs or alerts, this developer experience leaves a lot to be desired. We commonly burn a ton of time in the second step: after refreshing, we click through multiple extraneous screens before finally getting to the part of the application we're actually concerned about. We may make incremental progress each pass, but it may be repeated an inestimable number of times before we can call the bug fixed or the feature complete.

Let's start chipping away at these development-cycle times. Wouldn't it be nice if we no longer had to manually refresh the page after a change is made to our code? If we know we'll need to refresh the browser to view and test each code change, our build tools might as well take care of that for us.

Multiple tools are capable of this file-watching and updating on change, but we'll be referencing Webpack specifically throughout the rest of this chapter. Webpack is not required to use Redux, but it is a favorite among React developers and it comes already configured within apps generated by Create React App.

Webpack is a type of build tool — a module bundler — capable of all kinds of tasks, thanks to rich plugin options. In this chapter, however, we're interested in only those features that improve our Redux development workflow. Don't panic, no Webpack expertise is required to understand the concepts in this chapter.

So, Webpack can save us a second here and there with automatic refreshing. Not bad, but we're not that excited yet either. The next opportunity for optimization is to more quickly bundle changes into the application and perform that refresh. Turns out that's a specialty of Webpack. By omitting the resources not required by a component, Webpack allows us to send less over the wire for each page load. These optimizations come enabled out of the box with Create React App and require no further configuration.

The combination of automatic refreshes and faster load times are nice wins that may add up over time, but they're still just incremental improvements in a development workflow. You may be tempted to believe the Webpack bang isn't worth the buck, but we'll quickly discover that another feature, hot module replacement, is the foundation of an exceptional Redux development experience.

3.6 Hot module replacement

Hot module replacement enables an application to update without having to refresh. Let that sink in. If you've navigated deeply into an application to test a specific component, with hot module replacement, each code change updates in real time, leaving you to continue debugging uninterrupted. This feature all but eliminates that second, costly step in our example debugger workflow: "click through the application until triggering the debugger." There's Webpack giving you your money's worth.

Note that hot module replacement doesn't outright replace *debugger*. The two debugging strategies can be used harmoniously together. Use *debugger* no differently than you already do, and let hot module replacement reduce the time you might spend navigating to reach the same breakpoint in the following development cycle.

It's worth clarifying at this point that hot module replacement is a feature of Webpack and is not at all coupled or unique to React or Redux applications. Create React App enables the hot module replacement feature in Webpack by default, but it's still up to you to specify how to handle updates. There are two specific updates we'll want to configure for a React and Redux application. The first is how to handle updates to components.

3.6.1 Hot-loading components

Your next objective is to take Parsnip and augment it with hot module replacement. The first goal is to have Webpack update any components we touch without refreshing the page. Fortunately, the implementation logic is roughly that simple. See if you can make sense of the code in listing 3.2.

To summarize, Webpack will expose the `module.hot` object in development mode. One of the methods on that object is `accept`. `accept` takes two arguments: one or more dependencies and a callback. We'll want an update to any of our components to trigger the hot replacement, and fortunately, we don't have to list every React component in our application as a dependency. Whenever a child of the top-most component updates, the change will be picked up by the parent. We can just pass the string location of `App` to the `accept` function.

The second argument passed to `accept` is a callback that gets executed after the modules have successfully been replaced. In our case, we're just looking to render `App` and the rest of the updated components back to the DOM. In summary, each update to a component causes that module to be replaced, then those changes are rendered to the DOM without reloading the page.

Listing 3.2 src/index.js

```
…
if (module.hot) {                              ❶
   module.hot.accept('./App', () => {     ❷
      const NextApp = require('./App').default;
      ReactDOM.render(
         <Provider store={store}><NextApp /></Provider>,
         document.getElementById('root’)
      );
   });
}
```

1. Create React App has hot module replacement enabled in development mode
2. Whenever the App component (or one of its children) changes, re-render the component

Webpack will not expose `module.hot` in a production environment for good reason: we have no business making live changes to components in production. Remember that hot module replacement is a tool used only in development to accelerate development cycles.

3.6.2 Hot-loading reducers

It makes sense to add hot module replacement in one more location in the app: reducers. Manipulating data in reducers is another of those points in the development workflow that can really eat up the clock if you need to reload the page after each iteration. Consider instead the ability to make changes to a reducer and see data in their respective components update in real time.

In listing 3.3, we see a similar pattern to the implementation for components. In development mode, we listen for changes to our reducers and execute a callback after the modules have been replaced. The only difference now, is that instead of rendering new components to the DOM, we're replacing the old reducer with the updated one and recalculating what the state should be.

Listing 3.3 src/index.js

```
if (module.hot) {
   …

   module.hot.accept('./reducers’, () => {     ❶
      const nextRootReducer = require('./reducers’).default;
      store.replaceReducer(nextRootReducer);   ❷
   });
}
```

1. Whenever the reducer updates, perform the hot module replacement
2. The Redux store has a replaceReducer method to facilitate this update

Imagine that as you were developing the workflow for the `CREATE_TASK` action, you had misspelled `CREATE_TASK` in the reducer. You might've created several tasks while testing the code you were writing, and even seen the actions logged in the Redux DevTools, but no new tasks would appear in the UI. With hot module replacement applied to the reducer, the correction of that typo would result in the missing tasks appearing instantly — no refreshing required.

How is it that a change to a reducer can update the state of data already in the Redux store? The stage for this feature is set by the purity of the Redux architecture. We must be able to rely on the reducer function to be deterministic; the same action will always produce the same change in state. In other words, if the Redux store were mutable, we couldn't be certain that a list of actions would result in the same state each time.

Given a read-only store, this killer feature becomes possible with a clever combination of the Redux DevTools and hot module replacement. The short version of the answer is that the Redux Devtools augment the store with the ability to keep a running list of all the actions. When Webpack accepts an update to hot-load and calls `replaceReducer`, each of those actions is replayed through the new reducer. Presto! A recalculated state is born. This happens instantly and saves a ton of time having to re-create the same state manually.

Now we're cooking with fire! When developing, we can make changes to components or reducers and expect to see changes instantly, while maintaining the state of the application. You can imagine this saves quite a bit of development time, but the real *aha* moments come with experience. Try implementing hot module replacement for yourself before moving on to the next section.

3.6.3 Limitations of hot module replacement

Note that hot module replacement currently has a few limitations. Updating some non-component files may require a full page refresh, for example, and a console warning may tell you as much. The other limitation to be aware of is the inability to maintain local state in React components.

Remember, it's perfectly reasonable to use local component state in combination with the Redux store. Hot module replacement has no trouble leaving the Redux store intact, but maintaining local state after an update is a tougher puzzle to solve. When App and its children components are updated and re-rendered to the DOM, React sees these as new and different components, and they lose any existing local state in the process.

There is one tool that makes it possible to go that step further and maintain local component state after a hot module replacement: React Hot Loader.

3.7 Preserving Local State with React Hot Loader

React Hot Loader is another of Dan Abramov's pet projects and a version was demonstrated with Redux in his popular 2015 React Europe conference talk, "Hot Reloading with Time Travel." That early, experimental library has come a long way since then. Several iterations later, a stable package is available for use in your projects now.

As we've alluded, React Hot Loader takes the hot module replacement developer experience a step further. For every code update, not only will the Redux store be preserved, so too will each component's local state. React Hot Loader achieves this with the nuanced use of component proxies. Fortunately for us end users, those implementation details are hidden under the hood and a simple configuration is all that's required to enjoy the fruits of that labor.

One downside to React Hot Loader is the unfortunate incompatibility with Create React App. It requires some configuration of either Babel or Webpack, which necessitates ejecting (`npm run eject`) from Create React App. We will not be ejecting from the application in this book, so implementing React Hot Loader is left as an exercise for the reader. Please see the React Hot Loader GitHub repository at https://github.com/gaearon/react-hot-loader for instructions.

The value of adding React Hot Loader comes down to how large or how complex the local state in your application becomes. Many Redux applications rely on only simple local state to store of the contents of a form, before the user submits it, for example. In these cases, vanilla hot module replacement is generally more than sufficient for an excellent developer experience.

3.8 Exercise

As a quick exercise to get more familiar with the Redux DevTools, try navigating to the Chart Monitor to view a graphic visualization of your application's state.

3.9 Solution

This is a quick one; the solution is only two clicks away. Recall that in the upper-left corner of the Redux DevTools, you can click on the name of the current monitor to reveal a dropdown list of more available monitors.

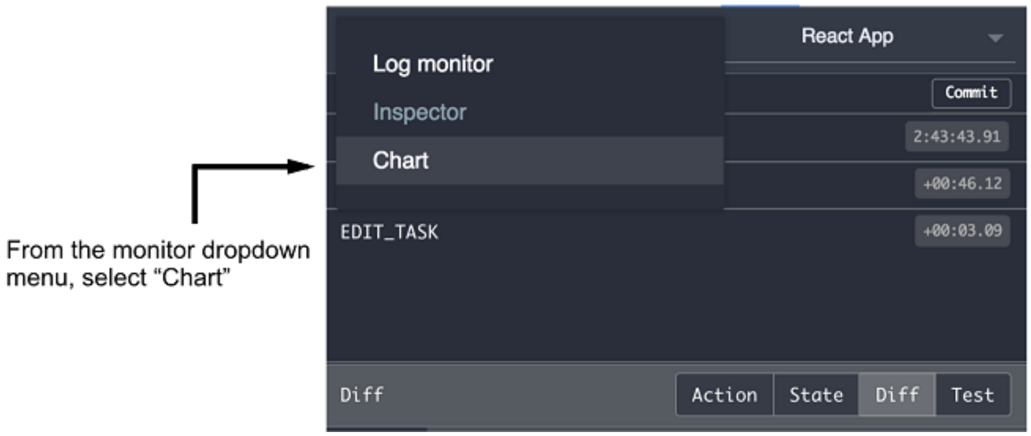
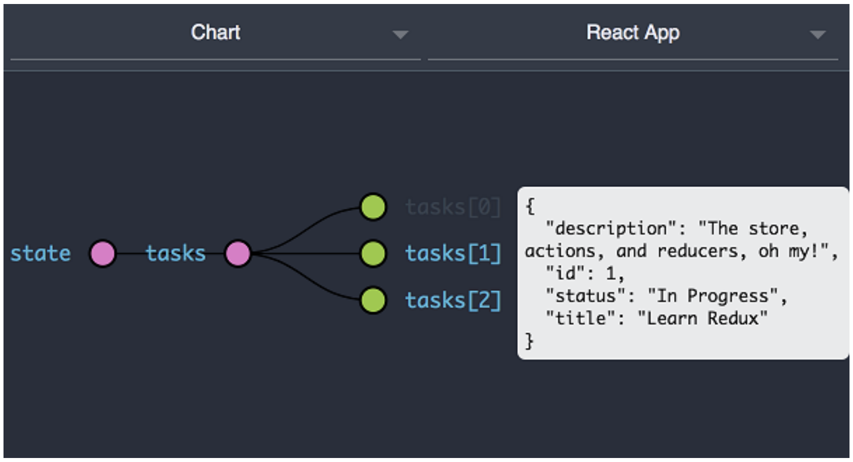


Figure 3.7 The location of the Chart Monitor

Clicking on the "Chart" option will reveal a graphical representation of the state in your application. It won't look so impressive now, because there's not a lot going on yet. You can imagine that a fully-featured application would contain a much larger web of data, though. Using this perspective won't always be the best lens on your data, but it has its moments.



```
{
  "description": "The store,
actions, and reducers, oh my!",
  "id": 1,
  "status": "In Progress",
  "title": "Learn Redux"
}
```

Figure 3.8 The Chart Monitor

Hovering over nodes in the Chart Monitor reveals their contents. This feature makes for a convenient way to quickly get to know an application. Navigation within the Chart Monitor can be controlled by zooming in and out, or clicking and dragging to make lateral movements.

3.10 Summary

Your investment in learning the debugging tools and strategies covered in this chapter will start to pay immediate dividends, getting you unstuck in upcoming examples or on your own projects. The Redux developer experience is second to none and that's thanks in large part to the Redux DevTools. In this chapter, we learned that:

- The Redux developer tools allow for the visualization and manipulation of actions in real time.
- Monitors determine how the data is visualized and the DevTools can mix and match monitors.
- Hot module replacement takes the development experience to new heights by performing updates without refreshing the page.

The next chapter is a great place to put these new debugging skills to the test, where we'll introduce asynchronous actions and interact with an API.

4 Consuming an API

This chapter covers

- **Asynchronous actions**
- **Handling errors with Redux**
- **Rendering loading states**

If you've kept up so far with the book, or completed a few of the more basic tutorials online, you know that we've reached a point where things traditionally start to get a little more tricky. Here's where we are: a user interacts with the app and actions get dispatched to reflect certain events, like creating a new task or editing an existing task. The data lives directly in the browser, which means we lose any progress if the user refreshes the page.

Perhaps without realizing, every action you've dispatched so far has been *synchronous.* When an action is dispatched, the store receives it immediately. These kinds of actions are straightforward, and synchronous code is generally easier to work with and think about. You know exactly the order things will execute based on where the code is defined. We execute line one and then line two, and so on. But you'd be hard-pressed to find any real-world JavaScript application that didn't involve some asynchronous code. It's fundamental to the language and required to do what this chapter is all about: talking with a server.

4.1Asynchronous actions

To repeat a few fundamental Redux ideas:

- Actions are objects that describe an event, like `CREATE_TASK`.
- Actions must be dispatched to apply any updates.

Every action we've dispatched in chapters 2 and 3 has been synchronous, but, as it turns out, we often need a server to do anything useful. Specifically, we need to make AJAX requests to do things like fetch tasks when the app starts and save new tasks to a server, so they'll persist between page refreshes. Virtually every real-world application is backed by a server of some kind, so clear patterns for interacting with an API are crucial for an application's long-term health.

Figure 4.1 recaps how our dispatches have looked so far. We dispatch an action from the view, and the action is received immediately by the store. All this code is synchronous, meaning each operation will run only after the previous operation has completed.

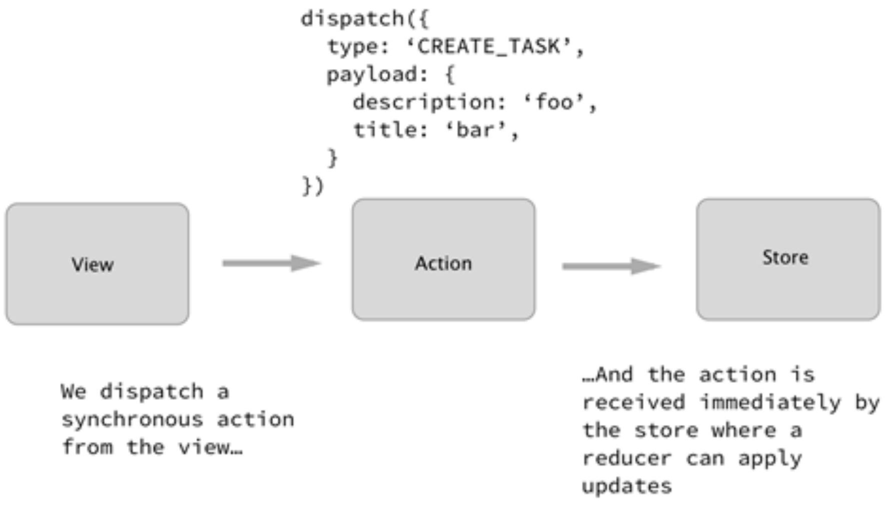

dispatch({
  type: 'CREATE_TASK',
  payload: {
    description: 'foo',
    title: 'bar',
  }
})
View
Action
Store
We dispatch a synchronous action from the view…
…And the action is received immediately by the store where a reducer can apply updates

Figure 4.1 A synchronous action being dispatched.

By contrast, asynchronous, or *async*, actions are where we'll be able to add asynchronous code like AJAX requests. Instead of dispatching a single synchronous action, async actions will typically have the following:

- One or more side-effects, like an AJAX request
- One or more synchronous dispatches, like dispatching an action after an AJAX request has resolved

In figure 4.2, notice the delay between the initial async action being dispatched, and when the store finally receives an action to process.

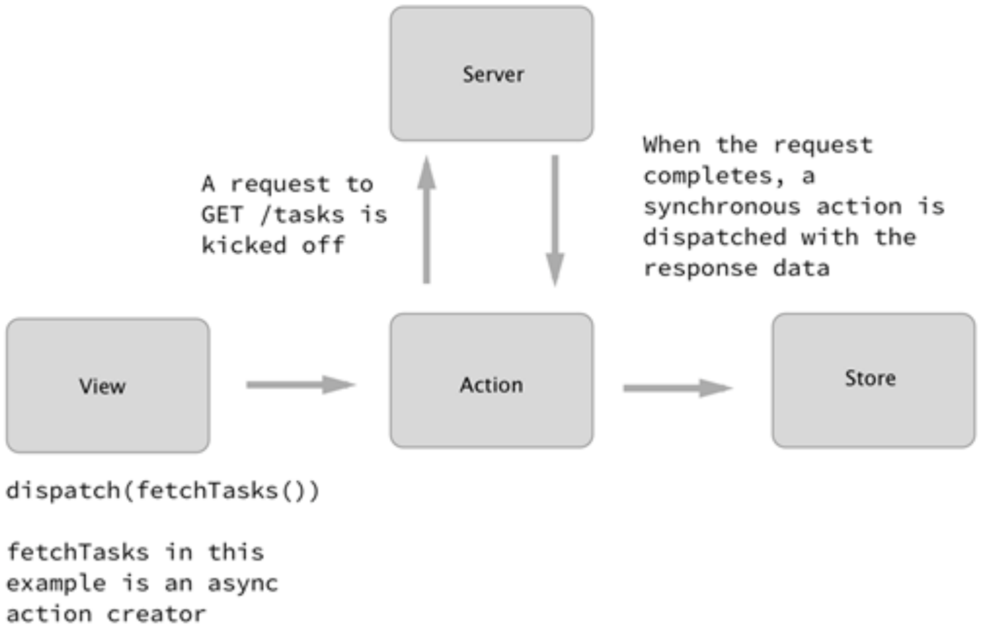

Server
A request to GET /tasks is kicked off
When the request completes, a synchronous action is dispatched with the response data
View
Action
Store
dispatch(fetchTasks())
fetchTasks in this example is an async action creator

Figure 4.2 An asynchronous action dispatch.

Much of the confusion around async actions boils down to terminology. In this chapter (and throughout the book), we'll try to be very specific about the language we use to avoid overloading certain language. Here are the fundamental concepts along with examples:

- Action – an object that describes an event. The phrase *synchronous action* always refers to these action objects. If a synchronous action is dispatched, it is received by the store immediately. Actions have a required `type` property, and can optionally have additional fields that store data need to handle the action.

```
{
  type: 'FETCH_TASKS_SUCCEEDED',
  payload: {
    tasks: [...]
  }
}
```

- Action creator – a function that returns an action.
- Synchronous action creator – all action creators that simply return an action are considered *synchronous* action creators.

```
function fetchTasksSucceeded(tasks) {
  return {
    type: 'FETCH_TASKS_SUCCEEDED',
    payload: {
      tasks: [...]
    }
  }
```

- Async action creator – an action creator that *does* contain async code (the most common example being network requests). As we'll see later in the chapter, they typically make one or more API calls and dispatch one or more actions at certain points in the request's lifecycle. Often, instead of returning an action directly, they may return synchronous action creators for readability.

```
export function fetchTasks() {
  return dispatch => {
    api.fetchTasks().then(resp => {
      dispatch(fetchTasksSucceeded(resp.data));
    });
  };
}
```

The syntax from the last example is a sneak peak at what's ahead. Soon we'll introduce the `redux-thunk` package, which will allow us to dispatch functions in addition to standard action objects. Before we can dig into implementing some async action creators, we're going to need a simple server. Please go to appendix A and follow the instructions for setting up that server and installing two more dependencies: `axios` and `redux-thunk`. Don't miss the important tweak required in the Redux DevTools configuration either! When you've finished, go on to the next section to learn how to dispatch async actions.

4.2Invoking async actions with redux-thunk

We know we can pass an action object to the `dispatch` function, which will pass the action to the store and apply any updates. What if we don't want the action to be processed immediately? What if we want to make a GET request for tasks and dispatch an action with the data from the response body? The first async action you'll dispatch will be to fetch a list of tasks from the server when the app loads. At a high level, you'll do the following:

- From the view, dispatch an asynchronous action to fetch tasks
- Perform the AJAX request to GET /tasks
- When the request completes, dispatch a synchronous action with the response

We've been going on about async actions, but we've yet to show you the mechanism to accomplish them quite yet. We can transition the `fetchTasks` action creator from synchronous to asynchronous by simply returning a function instead of an object. The function we return from `fetchTasks` can safely perform a network request and dispatch a synchronous action with some response data.

It is absolutely possible to do this without another dependency, but you'll find that the code to facilitate it may quickly become unmanageable. For starters, each of your components will need to be aware if they're making a synchronous or asynchronous call, and if the latter, pass along the dispatch functionality.

The most popular option for handling async actions is `redux-thunk`, a Redux *middleware*. Understanding the ins and outs of middleware isn't necessary now; we'll cover middleware in depth in chapter 5. The most important takeaway is that adding the `redux-thunk` middleware allows us to dispatch functions as well as the standard action objects that you're already used to. Within these functions, we're safe to add any asynchronous code we might need.

4.2.1 Fetching tasks from a server

At this point, we have a functioning HTTP API, an AJAX library (`axios`), and the `redux-thunk` middleware, which will allow us to dispatch functions instead of action objects when we need to perform async operations like network requests.

Currently, we're rendering the page with a static list of tasks defined in the `tasks` reducer. Start by removing the list of mock tasks in `src/reducers/index.js`, and adjust the initial state for the reducer.

Listing 4.3 src/reducers/index.js

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

const mockTasks = [  //#A
  {
    id: uniqueId(),
    title: 'Learn Redux',
    description: 'The store, actions, and reducers, oh my!',
    status: 'Unstarted',
  },
  {
    id: uniqueId(),
    title: 'Peace on Earth',
    description: 'No big deal.',
    status: 'In Progress',
  },
  {
    id: uniqueId(),
    title: 'Foo',
    description: 'Bar',
    status: 'Completed',
  },
];

export function tasks(state = { tasks: [] }, action) {  //#B
  ...
```

#A Remove the mockTasks array completely
#B Replace mockTasks with an empty array as the initial state for the tasks property

At a high-level, here's what you'll need to add to fetch a list of tasks via AJAX:

- When the app loads, dispatch an async action, `fetchTasks`, to fetch the initial tasks.
- Make the AJAX call to `/tasks`.
- When the request completes, dispatch a synchronous action `FETCH_TASKS_SUCCEEDED` with the result.

Figure 4.3 shows the async action we're about to create in more detail.

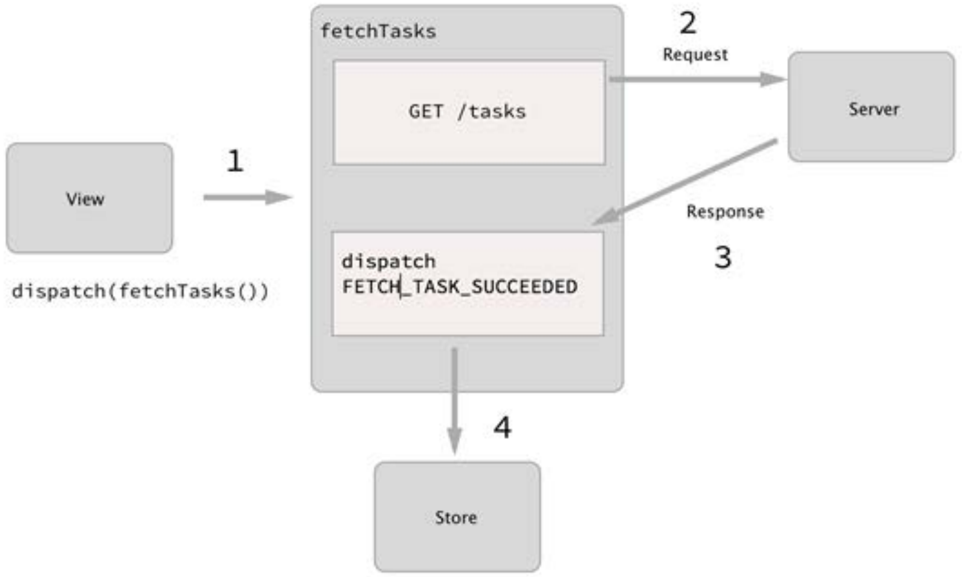


Figure 4.3 The chronological flow of fetchTasks as an async action.

Just like with synchronous actions, the goal of dispatching `fetchTasks` as an async action is to load some tasks into the store, so we can render them on the page. The only difference here is that we fetch them from a server instead of relying on mock tasks defined directly in the code.

Keeping figure 4.3 in mind, let's start from left to right by dispatching an action from the view. Import a soon-to-be-created `fetchTasks` action creator and dispatch it within the `componentDidMount` lifecycle method.

Listing 4.4 src/App.js

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import TasksPage from './components/TasksPage';
import { createTask, editTask, fetchTasks } from './actions';  //#A


class App extends Component {
  componentDidMount() {
    this.props.dispatch(fetchTasks()); //#B
  }

  ...

  render() {
    return (
      <div className="main-content">
        <TasksPage
          tasks={this.props.tasks}
          onCreateTask={this.onCreateTask}
          onStatusChange={this.onStatusChange}
        />
      </div>
    );
  }
}

function mapStateToProps(state) {
  return {
    tasks: state.tasks
  };
}
```

#A Import the `fetchTasks` action creator from the actions module
#B Dispatch the `fetchTasks` action from `componentDidMount`.

It's useful to think about apps in terms of data dependencies. We should be asking ourselves, "What data does this page or page section need to render successfully?" When we're working in the browser, `componentDidMount` is the appropriate lifecycle callback to initiate AJAX requests. Because it's configured to run whenever a component is first mounted to the DOM, it's at this point that we can begin to fetch the data to populate that DOM. Furthermore, this is an established best practice in React.

Next, head to `src/actions/index.js` where you'll need two things:

- An implementation of `fetchTasks` that performs the AJAX call.
- A new synchronous action creator, `fetchTasksSucceeded`, to dispatch the tasks from the server response into the store.
- Use axios to perform the AJAX request. Upon a successful response, the body of that response will be passed to the synchronous action creator to be dispatched.

Listing 4.5 src/actions/index.js

```
import axios from 'axios';

...

export function fetchTasksSucceeded(tasks) {  //#A
  return {
    type: 'FETCH_TASKS_SUCCEEDED',
    payload: {
      tasks
    }
  }
}

export function fetchTasks() {
  return dispatch => {  //#B
    axios.get('http://localhost:3001/tasks')  //#C
      .then(resp => {
        dispatch(fetchTasksSucceeded(resp.data));  //#D
      });
  }
}
```

#A A new synchronous action that will be dispatched if the request completes successfully
#B fetchTasks returns a function instead of an action
#C Make the AJAX request
#D dispatch a synchronous action creator

The biggest shift in Listing 4.4 from any of the actions we've worked with so far is that `fetchTasks` returns a function, not an action object. The `redux-thunk` middleware is what makes this possible. If we attempted to dispatch a function without the middleware applied, Redux would throw an error because it expects an object to be passed to `dispatch`.

Within this dispatched function, we're free to do the following:

- Make an AJAX request to fetch all tasks
- Dispatch a synchronous action with a result
- Most async actions tend to share these basic responsibilities.

4.2.2 API clients

While we let async actions start to sink in, let's cover a common abstraction for interacting with servers. In listing 4.5, we used `axios` for the first time to make a `GET` request to the `/tasks` endpoint. This is fine for the time being, but as our application grows we'll start to run into a few issues. What if we changed the base URL for the API from `localhost:3001` to `localhost:3002`? What if we wanted to use a different AJAX library? We have to update that code in only one place now, but imagine if we had ten AJAX calls. What about 100?

To address these questions, we can abstract those details into an API client and give it a friendly interface. If you're working with a team, future developers won't have to worry about specifics around ports, headers, the AJAX library being used, and so on.

Create a new `api/` directory with a single `index.js` file. If you're working in a large application with many difference resources, it can make sense to create multiple files per resource, but a single `index.js` file will be fine to get us started.

In `src/api/index.js`, create a `fetchTasks` function that will encapsulate your API call, and configure `axios` with basic headers and a base URL.

Listing 4.6 *src/api/index.js*

```
import axios from 'axios';

const API_BASE_URL = 'http://localhost:3001'; //#A

const client = axios.create({
  baseURL: API_BASE_URL,
  headers: {
    'Content-Type': 'application/json', //#B
  },
});
```

```
export function fetchTasks() { //#C
  return client.get('/tasks'); //#C
} //#C
```

#A Define a constant for the API's base URL
#B The `Content-Type` header is required by json-server for PUT requests.
#C Export a named `fetchTasks` function which will take care of making the call

Here we're hardcoding the base URL for the API. In a real-world application, you'd likely get this from a server, so the value can be different based on the environment, like staging or production.

With `fetchTasks`, we're encapsulating the request method as well as the URL for the endpoint. If either change, now we only have to update code in one place. Note that `axios.get` returns a promise, which we can call **.then** and `.catch` on from within an async action creator.

Now that you're exporting a `fetchTasks` function that wraps the API call, head back to `src/actions/index.js` and replace the existing API call with the new `fetchtasks` function.

Listing 4.7 src/actions/index.js

```
import * as api from '../api'; // #A

...

export function fetchTasks() {
  return dispatch => {
    api.fetchTasks().then(resp => {  //#B
      dispatch(fetchTasksSucceeded(resp.data));
    });
  };
}

...
```

#A Import all available API methods
#B Use the friendlier interface for making an AJAX call

Not only are the details of the request safely hidden away, the `fetchTasks` action creator is also more clear and concise. By extracting an API client, we've improved encapsulation, future maintainability, and readability at the cost of the overhead of another module to manage. Creating new abstractions isn't always the right answer, but in this case, it seems like a no-brainer.

4.2.3 View and server actions

We now know a few things about synchronous and asynchronous actions, but there's one more concept that can help us form a clear picture around how updates are happening in our applications. There are typically two entities that can modify application state: users and servers. Actions can be divided into two groups, one for each actor: *view actions* and *server actions*.

- View actions are initiated by users. Think `FETCH_TASKS`, `CREATE_TASK`, and `EDIT_TASK`. For example, a user clicks a button and an action is dispatched.
- Server actions are initiated by a server. For example, a request successfully completes, and an action is dispatched with the response. When we implemented fetching tasks via AJAX we introduced our first server action: `FETCH_TASKS_SUCCEEDED`.

NOTE **Some developers like to organize view and server actions in separate directories, the argument being it can help to break up larger files. That being said, it's not a requirement and we won't do so in this book.**

Getting back to the code, we have a server action, `FETCH_TASKS_SUCCEEDED`, which we dispatch with the list of tasks sent back from the server. Server actions are initiated by a server event like a response, but they still behave just like any other action. They get dispatched and then handled by a reducer.

Let's wrap up our initial fetching logic by updating the tasks reducer to handle receiving tasks from the server. It's also safe to remove the `mockTasks` array at this point. Because we're able to remove the mockTasks array, we can use an empty array as the initial state for the reducer.

Listing 4.8 src/reducers/index.js

```
...
export default function tasks(state = { tasks: [] }, action) {  // #A

  ...

  if (action.type === 'FETCH_TASKS_SUCCEEDED') { // #B
    return {
      tasks: action.payload.tasks,
    };
  }

  return state;
}
```

#A Make sure to pass an empty array as the initial state for `tasks`
#B The reducer now listens for the server action

Notice how we didn't have to make any updates to the view? That's by design! Our React components don't particularly care *where* the tasks come from, which allows us to totally change our strategy for acquiring tasks with relatively low effort. Keep this in mind as we continue to move through the book. When possible, we should be building each piece of our app with a clear interface – changing one piece (such as how we get a list of tasks to initially render) shouldn't affect another (the view).

At this point we've introduced most of the conceptual heavy lifting that chapter 4 has to offer, so let's do a quick recap before we move on to persisting new tasks to the server. We've covered the following:

- Asynchronous actions and `redux-thunk`. `redux-thunk` allows us to dispatch functions instead of objects, and inside those functions we can make network requests and dispatch additional actions when any requests complete.

- Synchronous actions. Dispatching an action object with a `type` and `payload` is considered a synchronous action, because the store receives and processes the action immediately after dispatch.

Users and servers are the two actors that can modify state in our applications. As a result, we can group actions into view actions and server actions.

4.3Saving tasks to the server

Now that we're fetching tasks from the server when the app loads, let's update creating tasks and editing tasks to be persisted on the server. The process will be very similar to what we've seen so far with fetching tasks.

Let's start by saving new tasks. We already have a framework in place: when a user fills out the form and submits, we dispatch the `createTask` action creator, the store receives the `CREATE_TASK` action, the reducer handles updating state, and the changes are broadcast back to the UI.

`createTask` needs to return a function instead of an object. Within that function, we can make our API call and dispatch an action when a response is available. Here's a quick look at the high-level steps:

- Convert the synchronous `createTask` action creator into an async action creator.
- Add a new method to our API client, which will send a POST request to the server.
- Create a new server action, `CREATE_TASK_SUCCEEDED`, whose payload will be a single task object.

In the `createTask` action creator, initiate the request, and dispatch `CREATE_TASK_SUCCEEDED` when the request finishes. For now, we can assume it will always be successful.

Remove the `uniqueId` function. It was originally meant as a stopgap until we could create tasks on a server, which would be responsible for adding an ID.

Now we're there! Create a new function for creating tasks in the API client.

Listing 4.8 src/api/index.js

```
...
export function createTask(params) {
  return client.post('/tasks', params); // #A
}
```

#A A POST request is required to add or update data on the server

Now we can modify the `createTask` action creator to return a function.

Listing 4.9 src/actions/index.js

```
import * as api from '../api';
```

```
...

function createTaskSucceeded(task) { //#A
  return {
    type: 'CREATE_TASK_SUCCEEDED',
    payload: {
      task,
    },
  };
}

export function createTask({ title, description, status = 'Unstarted' }) {
  return dispatch => {
    api.createTask({ title, description, status }).then(resp => {
      dispatch(createTaskSucceeded(resp.data));  //#B
    });
  };
}
```

#A Create a new synchronous action creator
#B Load the newly created object into the store

We know that reducers handle updates to state, so update the tasks reducer to handle the `CREATE_TASK_SUCCEEDED` action. After that, we'll be up to four action handlers, so now is as good a time as any to merge each **if** statement into a friendlier `switch` statement. This is a common Redux pattern.

Listing 4.10 src/reducers/index.js

```
export default function tasks(state = { tasks: [] }, action) {
  switch (action.type) {  //#A
    case 'CREATE_TASK': {
      return {
        tasks: state.tasks.concat(action.payload),
      };
    }
    case 'EDIT_TASK': {
      const { payload } = action;
      return {
        tasks: state.tasks.map(task => {
          if (task.id === payload.id) {
            return Object.assign({}, task, payload.params);
          }

          return task;
        }),
      };
    }
    case 'FETCH_TASKS_SUCCEEDED': {
      return {
        tasks: action.payload.tasks,
      };
    }
    case 'CREATE_TASK_SUCCEEDED': {  //#B
      return {
        tasks: state.tasks.concat(action.payload.task),
      };
    }
    default: {
      return state;
    }
  }
}
```

#A Move to a switch statement instead of a long if-else chain
#B The new action handler

A switch statement is a slightly friendlier syntax when there are a significant number of cases to handle. This is the structure that you'll typically see most often in Redux reducers, but using a switch statement isn't a hard requirement.

Play around in the browser and create some new tasks. When you refresh, they should appear again for the initial render. The big idea here is that the `createTask` action creator now returns a function instead of an object. The newly created task isn't received by the store immediately after being dispatched, but instead is dispatched to the store after the **POST** request to `/tasks` has completed.

4.4 Exercise

Task updates are the last feature we need to hook up to the server. The process for fetching, creating, and now editing tasks is nearly identical. This exercise is a good way to test whether you're ready to connect the dots on your own.

We'll outline the high-level steps needed to make everything work, but as a challenge, see if you can implement the code before glancing through the listings in the solution section. The requirements are the following:

- Add a new API function for updating tasks on the server.
- Convert the `editTask` action creator from synchronous to asynchronous.
- Within `editTask`, kick off an AJAX request.
- When the request is complete, dispatch an action with the updated object that comes back as part the server response.

4.5 Solution

The first thing we'll look to do is add a new API function, `editTask`.

Listing 4.11 src/api/index.js

```
export function editTask(id, params) {
  return axios.put(`${API_BASE_URL}/tasks/${id}`, params); // #A
}
```

#A Use an ES2015 template string to easily construct the URL

Now that you have a function you can import to make the right AJAX request, create a new async action creator to make the request to the server, and a new synchronous action creator to indicate the request has completed.

Listing 4.12 src/actions/index.js

```
...

function editTaskSucceeded(task) { //#A
  return {
    type: 'EDIT_TASK_SUCCEEDED',
    payload: {
      task,
    },
  };
}

export function editTask(id, params = {}) {
  return (dispatch, getState) => {
    const task = getTaskById(getState().tasks.tasks, id); //#B
    const updatedTask = { //#B
      ...task, //#B
      ...params, //#B
    }; //#B
    api.editTask(id, updatedTask).then(resp => {
      dispatch(editTaskSucceeded(resp.data));
    });
  };
}

function getTaskById(tasks, id) {
  return tasks.find(task => task.id === id);
}
```

#A Create a new synchronous action creator for edits
#B Merge the new properties into the existing task objects

For each action that requires a network request (meaning we're dealing with an async action) we'll need at least one synchronous action creator to indicate where we are in the request/response lifecycle. Here, that's `editTaskSucceeded`, which indicates the request has completed successfully and passes data from the response body on to the reducer.

Because `json-server` requires a full object to be passed along for PUT requests, we have to grab the task out of the store and merge in the new properties ourselves.

Listing 4.13 src/reducers/index.js

```
export default function tasks(state = { tasks: [] }, action) {
  switch (action.type) {

    ...

    case 'EDIT_TASK_SUCCEEDED': { // #A
      const { payload } = action;
      return {
        tasks: state.tasks.map(task => {
          if (task.id === payload.task.id) { //#B
            return payload.task;
          }

          return task;
        }),
      };
    }
    default: {
      return state;
    }
  }
}
```

#A Handle the new server action
#B Replace the old task with the updated one

As you may have noticed, the process for saving updates to tasks is very similar to the process for creating tasks. All interactions that trigger an async operation (usually a network request) tend to have these same high-level happenings:

- A user interacts with the UI in some way, triggering a dispatch.
- A request is started.
- When the request finishes, an action is dispatched with some response data.

Go ahead and update a few tasks. The UI should be responsive given that our server is running locally, and the work that the server is doing is inexpensive. But that won't always be the case. Requests in real-world applications inevitably will take longer due to latency or expensive operations, which means we need some kind of user feedback while a request is completing. Segue!

4.6 Loading states

As UI programmers, we want to always keep our users well-informed of what's happening in the application. Users have some expectation that certain things take time to complete, but they won't forgive being left in the dark. When creating user experiences, user confusion should be one of the primary things we try and eliminate completely.

Enter loading states! We'll use Redux to track the status of a request, and update the UI to render the proper feedback when a request is in progress. One obvious place to start is during the initial fetch for tasks when the page loads.

4.6.1 The request lifecycle

With network requests, there are two moments in time that we care about: when the request starts, and when it completes. If we model these events as actions, we end up with three distinct action types that help describe the request-response lifecycle. Using fetching tasks as an example, note the following are the three action types:

- `FETCH_TASKS_STARTED` – dispatched when a request is initiated. Typically used to render a loading indicator (which we'll do in this section).

- `FETCH_TASKS_SUCCEEDED` – dispatched when a request is completed successfully. Takes some data from the response body and loads it into the store.
- `FETCH_TASKS_FAILED` – dispatched when a request fails for any reason, such as a network failure, or a response with a non-200 status code. Payloads often include an error message from the server.

Right now, the `fetchTasks` action creator accounts for only one moment in time, when the request completes. Figure 4.4 is what `fetchTasks` might look like if we also wanted to track when the request was initiated as well.

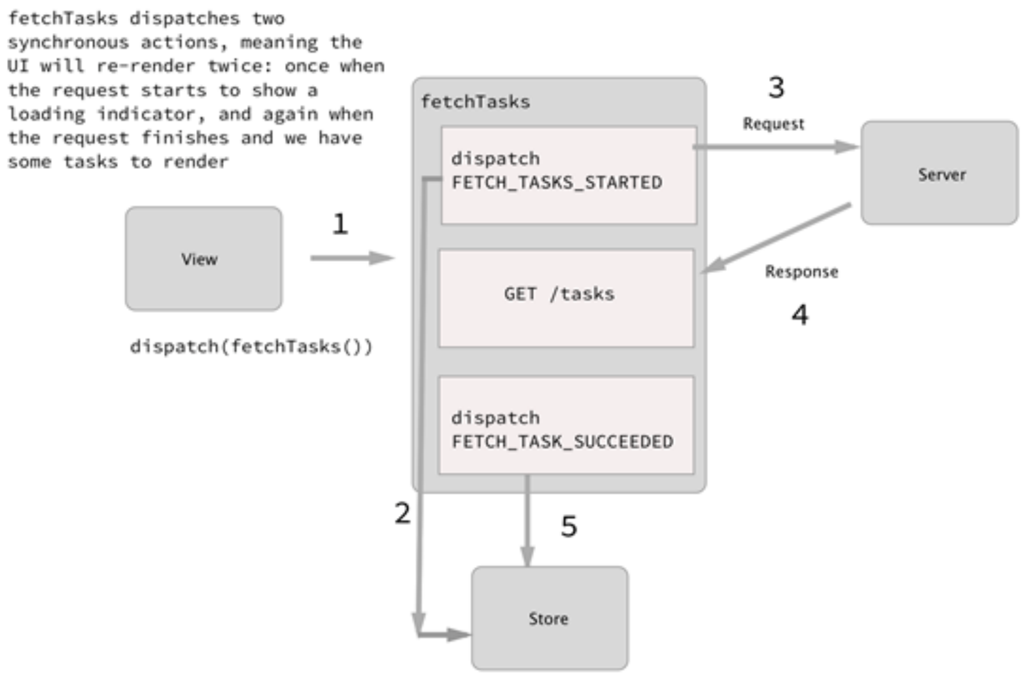

fetchTasks dispatches two synchronous actions, meaning the UI will re-render twice: once when the request starts to show a loading indicator, and again when the request finishes and we have some tasks to render
View
1
dispatch(fetchTasks())
fetchTasks
dispatch FETCH_TASKS_STARTED
GET /tasks
dispatch FETCH_TASK_SUCCEEDED
3
Request
Server
Response
4
2
5
Store

Figure 4.4 The fetchTasks async action creator with support for loading states.

Now, our async action creator, `fetchTasks`, will be responsible for three things:

- Dispatching an action to signify a request has been started
- Performing the request
- Dispatching a second action with some response data when the request completes

Think of async actions as orchestrations. They typically perform several individual tasks in pursuit of fulfilling some larger goal. Our goal is to fetch a list of tasks, but it takes several steps to get there. That’s the role of the `fetchTasks` async action creator: to kick off everything that needs to get done to accomplish that goal.

Our store has only a single property, tasks:

```
{
  tasks: [...]
}
```

To render a loading indicator when a request for tasks is in progress, we need to keep track of more state. Here’s how we want the store to look to track request status:

```
{
  tasks: {
    isLoading: false,
    tasks: [...]
  }
}
```

This structure is much more flexible, because it allows us to group the task objects along with any kind of additional state or metadata. In any real-world application, you’d be much more likely to come across a structure like this. The specific naming convention of the keys is just one pattern you can choose to adopt, but you're not obligated to.

4.6.2 Adding the loading indicator

We have a few things to take care of in the code to move toward this new state structure. To start, update the `tasks` reducer to take a new initial state, and update the existing action handlers.

Listing 4.13 src/reducers/index.js

```
const initialState = { //#A
  tasks: [], //#A
  isLoading: false, //#A
}; //#A

export default function tasks(state = initialState, action) {
  switch (action.type) {
    case 'FETCH_TASKS_SUCCEEDED': {
      return {
        ...state, //#B
        isLoading: false, //#B
        tasks: action.payload.tasks, //#B
      };
    }
    case 'CREATE_TASK_SUCCEEDED': {
      return {
        ...state, //#C
        tasks: state.tasks.concat(action.payload.task),
      };
    }
    case 'EDIT_TASK_SUCCEEDED': {
      const { payload } = action;
      const nextTasks = state.tasks.map(task => {
        if (task.id === payload.task.id) {
          return payload.task;
        }

        return task;
      });
      return {
        ...state, //#C
        tasks: nextTasks,
      };
    }
    default: {
      return state;
    }
  }
}
```

#A Define the new initial state for the reducer
#B Return the next state with the list of tasks from the payload
#C Include any existing state when updating the list of tasks

First, we made sure to set the `isLoading` flag to false by default when we defined the initial state for the reducer. This is always good practice, as it prevents any loading indicators from rendering when they're not supposed to. Let other actions that indicate a request has started set a flag like this to `true`.

When we handle the `FETCH_TASKS_SUCCEEDED` action, the obvious change is to update the array of tasks. We also have to remember to indicate the request is complete, so that any loading indicators are hidden, which we did by toggling the `isLoading` flag to false.

There's one important update we'll need to make all the way up in the creation of the store, in `index.js`. Previously, we were passing the tasks reducer directly to `createStore`. This was fine when all we had in our state object was a single array of tasks, but now we're moving toward a more complete structure with additional state that lives alongside the task objects themselves.

Create a small *root reducer* that takes the entire contents of the store (`state`) and the action being dispatched (`action`), and passes only the piece of the store that the tasks reducer cares about, `state.tasks`.

Listing 4.14 index.js

```
...
import tasksReducer from './reducers';

const rootReducer = (state = {}, action) => { //#A
  return {
    tasks: tasksReducer(state.tasks, action), //#B
  };
};

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

#A A rootReducer function that accepts the current state of the store and an action
#B Pass the tasks data and the action being dispatched to the tasks reducer

Adding a root reducer like this sets us up for the future as well. As we add more features to Parsnip, and as a result, have more data to track in Redux, we can add new top-level properties to the store and create reducers that operate only on relevant data. Eventually we'll add the ability for users to have different projects, and each project will have its own tasks. The top level of the redux store might look like this:

```
{
  tasks: {...},
  projects: {...}
}
```

To configure the store, all we'd need to do is add a line of code to the root reducer:

```
const rootReducer = (state = {}, action) => {
  return {
    tasks: tasksReducer(state.tasks, action),
    projects: projectsReducer(state.projects, action)
  };
};
```

This allows each reducer to not care about the overall shape of the store, only the slice of data that they operate on.

> **NOTE** **Using a root reducer like this is so common that redux exports a `combineReducers` function, which accomplishes the same thing as the `rootReducer` function we just wrote. Once we add a few more properties to our state object, we'll switch to the more standard `combineReducers`, but for now it's worth understanding how this process works under the hood.**

Next, run through the following familiar process of adding a new action by updating the relevant action, reducer, and component:

- Add and dispatch a new synchronous action creator `fetchTasksStarted`.
- Handle the `FETCH_TASKS_STARTED` action in the tasks reducer.

Update the `TasksPage` component to render a loading indicator when the fetch is in progress.

First, dispatch the new action. For dramatic effect, we'll add a `setTimeout` of two seconds before we dispatch `fetchTasksSucceeded`, which indicates the request has completed. Because the server responds almost instantly when running on our local machines, this delay gives us a chance to get a good feel for the loading state.

Listing 4.15 src/actions/index.js

```
function fetchTasksStarted() {
  return {
    type: 'FETCH_TASKS_STARTED',
  };
}

export function fetchTasks() {
  return dispatch => {
    dispatch(fetchTasksStarted()); //#A

    api.fetchTasks().then(resp => {
      setTimeout(() => { //#B
        dispatch(fetchTasksSucceeded(resp.data)); //#B
      }, 2000); //#B
    });
  };
}
```

#A Dispatch the fetchTasksStarted action creator to signify a request is in progress
#B setTimeout ensures the loading indicator will stay on the page for more than a fraction of a second

Now as part of our async action we have two synchronous dispatches to track the request lifecycle. One indicates when the request starts, the other when it completes. Next, handle the `FETCH_TASKS_STARTED` action by setting the `isLoading` property to true in the reducer.

Listing 4.16 src/reducers/index.js

```
...
export default function tasks(state = initialState, action) {
  switch (action.type) {
    case 'FETCH_TASKS_STARTED': {
      return {
        ...state,
        isLoading: true, //#A
      };
    }
    ...
  }
}
```

#A Setting the isLoading flag to true, which we'll eventually use in a React component to conditionally render a loading indicator.

Finally, we'll update two components, `App` and `TaskPage`. In `App`, pass the value of `isLoading` to the `TasksPage` via `mapStateToProps`.

Listing 4.17 src/App.js

```
...
class App extends Component {
  ...
  render() {
    return (
      <div className="main-content">
        <TasksPage
          tasks={this.props.tasks}
          onCreateTask={this.onCreateTask}
          onStatusChange={this.onStatusChange}
          isLoading={this.props.isLoading} //#A
        />
      </div>
    );
  }
}
function mapStateToProps(state) {
  const { tasks, isLoading } = state.tasks; //#B
  return { tasks, isLoading };
}
...
```

#A Passing the isLoading prop down to the TasksPage.
#B Updating mapStateToProps to pull isLoading out of the store and pass it to App as a prop.

Finally, in `TasksPage`, we can check whether a request for tasks is in progress, then render a loading indicator if necessary.

Listing 4.18 src/components/TasksPage.js

```
class TasksPage extends Component {
  ...
  render() {
    if (this.props.isLoading) {
      return (
        <div className="tasks-loading">
          Loading... //#A
        </div>
      );
      ...
    }
  }
```

#A Add a fancy loading animation here

Tracking the status of a request to show a loading indicator isn't necessarily required, but it's become part of the expected experience in most modern web apps. This is just one example of the increasingly dynamic requirements of modern web apps, but Redux was created in part to solve exactly these kinds of problems.

4.7 Error handling

With network requests, there are two moments in time we care about: the request starting, and the request finishing. The invoking of the `fetchTasks` request will always be represented by the `FETCH_TASKS_STARTED` action, but request completion can trigger one of two dispatches. We've handled only the success case so far, but proper error handling is crucial for delivering the best experience possible.

Users are never thrilled when something goes wrong, but having an operation fail and being left in the dark is much worse than seeing an error with some feedback. Taking one last look at the async action diagram in figure 4.5, the second dispatch can now fork depending on the outcome of the request.

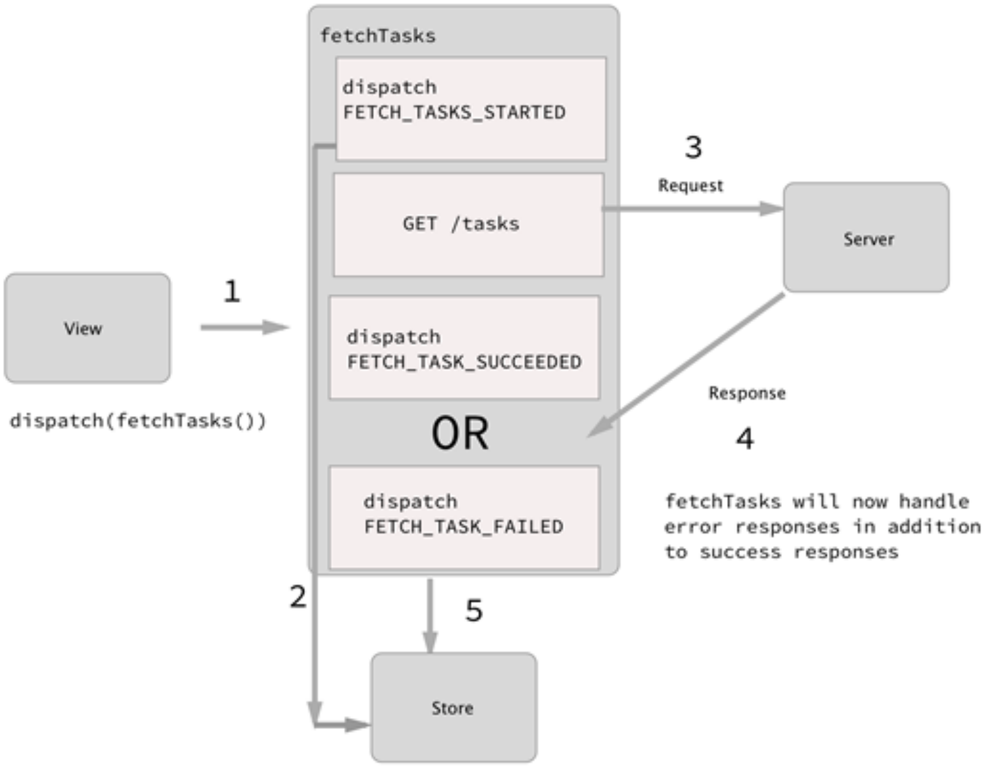

fetchTasks
dispatch FETCH_TASKS_STARTED
GET /tasks
dispatch FETCH_TASK_SUCCEEDED
OR
dispatch FETCH_TASK_FAILED
3
Request
Server
View
1
dispatch(fetchTasks())
Response
4
fetchTasks will now handle error responses in addition to success responses
2
5
Store

Figure 4.5 fetchTasks now handles the three types of actions that describe a request.

4.7.1 Dispatching an error action

There are plenty of ways to implement error handling in Redux. At a high level, these are the things you'll need:

- An action that dispatches an error message
- Somewhere to store the error message in the Redux store
- A React component to render the error

To keep things simple, we'll dispatch a single action, `FETCH_TASKS_FAILED`, with the error payload. When an error is present, we'll render the message at the top of the page, which will look something like figure 4.6.

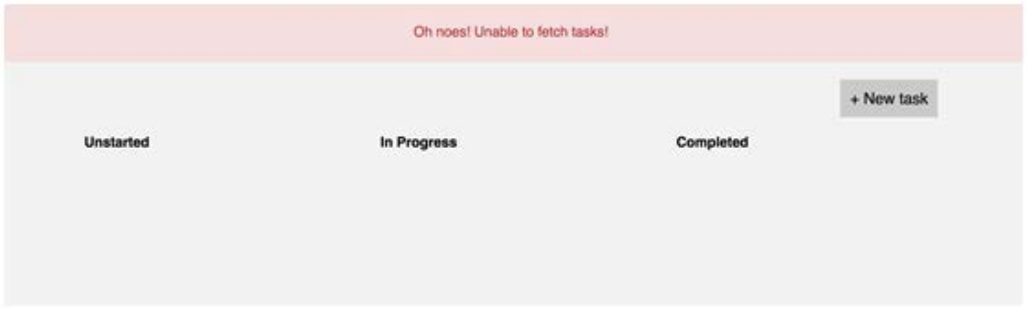


Figure 4.6 Rendering an error message.

Start from the outside-in by creating a new file for a `FlashMessage` component. Its purpose is to simply accept an error message as a prop and display it in the DOM.

Listing 4.19 src/components/FlashMessage.js

```
import React from 'react'; //#A

export default function FlashMessage(props) {
  return (
    <div className="flash-error">
      {props.message}
    </div>
  );
}

Error.defaultProps = {
  message: 'An error occurred', // #A
};
```

#A Even though we don't reference React directly in this file, the React object needs to be in scope to use JSX
#B A default error message is set

In the `App` component, pass along the yet-to-be-created `error` property from the redux store in `mapStateToProps`.

Listing 4.20 src/App.js

```
...
import FlashMessage from './components/FlashMessage';

class App extends Component {
  ...
  render() {
    return (
      <div className="container">
        {this.props.error &&
          <FlashMessage message={this.props.error} />} //#A
        <div className="main-content">
```

```
          <TasksPage
            tasks={this.props.tasks}
            onCreateTask={this.onCreateTask}
            onStatusChange={this.onStatusChange}
            isLoading={this.props.isLoading}
          />
        </div>
      </div>
    );
  }
}

function mapStateToProps(state) {
  const { tasks, isLoading, error } = state.tasks; //#B
  return { tasks, isLoading, error }; //#B
}

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

#A Conditionally rendering the FlashMessage component.
#B More mapStateToProps logic for passing data from the store into React.

Because `this.props.error` is `null` for now, nothing will happen quite yet in the UI. We'll need to create a new synchronous action creator, `fetchTasksFailed`. There is already code to handle when the request promise resolves successfully, so go ahead and add a catch block to handle when the promise is rejected.

To make testing error handling easier, manually reject a promise in the `then` block, so that you're guaranteed to make it into the `catch` block.

Listing 4.21 src/actions/index.js

```
function fetchTasksFailed(error) {
  return {
    type: 'FETCH_TASKS_FAILED',
    payload: {
      error,
    },
  };
}

export function fetchTasks() {
  return dispatch => {
    dispatch(fetchTasksStarted());

    api
      .fetchTasks()
      .then(resp => {
        // setTimeout(() => { //#A
        //   dispatch(fetchTasksSucceeded(resp.data)); //#A
        // }, 2000); //#A
        return Promise.reject(new Error('Oh noes! Unable to fetch tasks!')); //#B
      })
      .catch(err => {
        dispatch(fetchTasksFailed(err.message)); //#C
      });
  };
  ...
```

#A Comment out the success handler for now
#B Manually reject the promise, to test the code in the catch block
#C Dispatch another synchronous action with an error message

Finally, handle the update logic in the tasks reducer. This is a two-part change: add an error property to the initial state definition, then add a handler for the `FETCH_TASKS_FAILED` action. The case statement will mark the request as complete by setting `isLoading` to false and set the error message.

Listing 4.22 src/reducers/index.js

```
const initialState = {
  tasks: [],
  isLoading: false,
  error: null, //#A
};

export default function tasks(state = initialState, action) {
  switch (action.type) {
    ...
    case 'FETCH_TASKS_FAILED': { //#B
      return { //#B
        ...state, //#B
        isLoading: false, //#B
        error: action.payload.error, //#B
      };
    }
    ...
    default: {
      return state;
    }
  }
}
```

#A Set error to be null by default
#B Indicate the request is complete by setting the isLoading flag and error value

All said and done, it's clear that fetching a list of tasks to render on the page is much more than just making a GET request. These are the realities of modern web app development, but tools like Redux are here to help. We can handle tracking complex state to provide the best user experiences possible.

4.8 Summary

When first learning Redux, the stereotypical first tutorial is a todo app. It all seems so simple! Dispatch an action, update state in a reducer. But the question quickly turns to "How do we do anything useful?" Turns out we can't do much in a web application without being backed by a server.

Async actions are one of the real challenges for a budding Redux user. Compared to chapter 2, where only synchronous actions are found, the complexity level in this chapter was ramped up significantly. Hopefully you now have a sense for how to properly handle asynchronous code in Redux.

Throughout this chapter, you learned the following:

- The difference between dispatching asynchronous and synchronous actions
- How `redux-thunk` enables the dispatching of functions, which can be used to perform side effects, like network requests
- How API clients can reduce duplication and improve reusability
- The two conceptual groups of actions: view actions and server actions
- The three important moments during the lifecycle of a remote API call: start, successful completion, and failure
- Rendering errors to improve overall user experience

By using `redux-thunk`, you took advantage of some middleware without needing to understand what that meant. The next chapter will lift the veil on middleware and show you all there is to see.

5 Middleware

This chapter covers

- **What Redux middleware is**
- **How to write your own middleware**
- **Composing middleware**
- **When and when not to use middleware**

We’ve covered most of the usual suspects you’d find in a React/Redux application – actions, reducers, and the store. To update some state in your application via Redux, you need all three. There’s one more core actor that’s key to this whole operation – *middleware*. If you’ve spent any time in chapter 4, you've already come face-to-face with middleware and lived to tell the tale. When we added `redux-thunk` to the Parsnip project, we learned how to apply middleware to Redux via the `applyMiddleware` function, but not necessarily how to create our own. In this chapter, we’ll look more in depth at how middleware works, how to create it, and what use cases it can be a good fit for.

In the process, we’ll improve Parsnip by creating custom middleware for a few classic uses cases:

- Logging actions, which give us a quick look into what’s happening in the app
- Analytics, which provide a convenient interface to track an event when an action is dispatched
- API calls, which will abstract away common tasks around making calls to the server

Let’s get started!

5.1 What's in middleware?

What exactly is middleware? The concept isn't unique to Redux, and if you've worked with frameworks like Rails, Express.js, or Koa, you've likely used or been exposed to middleware in some shape or form. Generally, middleware is any code that runs between two other software components, typically as part of a framework. With a web server framework like Express or Koa, you can add middleware that runs after an incoming request is received, and before the framework handles a request. This is useful for all kinds of things, such as logging data about each request and response, handling errors in a centralized way, authenticating users, and so on.

Frameworks like Express and Koa are great for illustrating middleware, but we're here for Redux. If we know that middleware can be described as code that sits between two components, like a framework receiving an HTTP request and generating a response, where does middleware in Redux live?

Redux middleware is code that sits between an action being dispatched, and the store passing the action to the reducer and broadcasting the updated state. Just like server middleware is useful for running code across many requests, Redux middleware lets us run code across many action dispatches.

Let's look again at our architecture diagram (figure 5.1). The middleware section is highlighted to give you an idea of where this kind of code fits in to the normal action dispatch cycle.

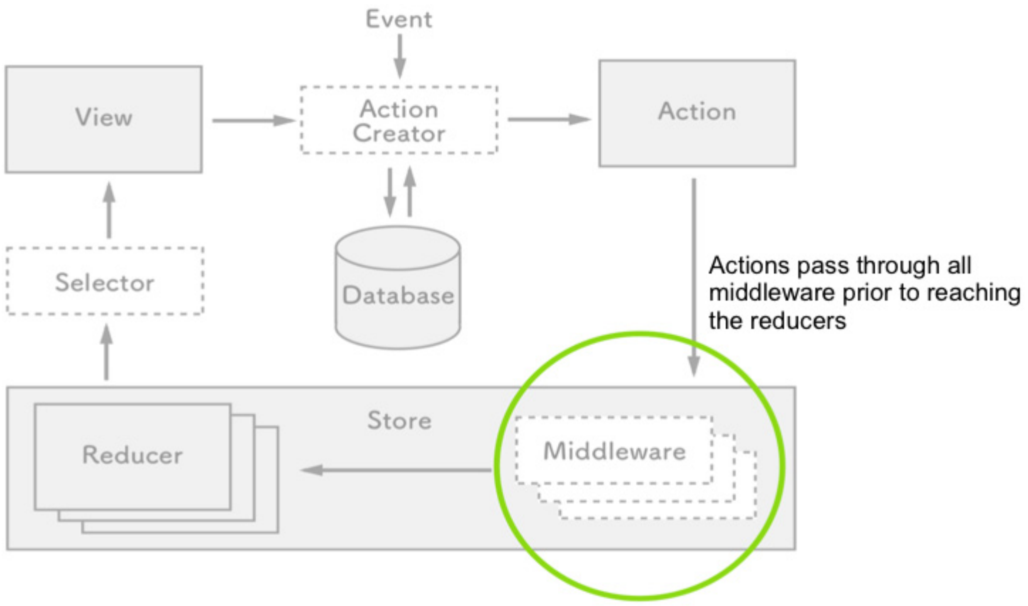


Figure 5.1 Actions move through middleware, then reducers prior to the final state being calculated.

Notice that middleware in figure 5.1 exists within the store. Think of middleware as being "registered" with the store. When an action is dispatched, the store will know to pass that action through any middleware you've added. When the entire chain of middleware is complete, the action is finally passed to the reducers to calculate the updated state of the application.

5.2 Middleware basics

Creating middleware involves a few basic steps:

- Define the middleware using the correct function signature
- Register the middleware with the store using Redux's `applyMiddleware` function

Let's start with the function definition. Here's the function signature for Redux middleware:

```
const middlewareExample = store => next => action => { ... }
```

Not confusing at all, right? In short, this amounts to writing three nested functions for each new middleware we write. At this stage in the game, it is most important to know the nature of the arguments provided to your middleware:

- `store` – the redux store. You can use the store object directly in middleware when you need to make decisions based on some existing state. `store.getState` has you covered.
- `next` – a function which you'll call when you're ready to pass the action on to the next middleware in the chain.
- `action` – the action being dispatched. Generally, your middleware will do something with every action (like logging), or watch for a specific action by checking the value of `action.type`.

5.2.1 Composing middleware

One of the key aspects of middleware is its ability to be chained, meaning multiple middleware can be applied to the store. Each middleware, after completing any work it may decide to do, will invoke the next middleware in the chain. As a result, any middleware you create should be focused and have a single purpose, making them easier to combine and reuse in different contexts. Maybe another app in your ecosystem can use a middleware you've created, or maybe you can open-source it! Because all Redux middleware must be created the same way, it's perfectly fine (and expected) to combine your own middleware with third-party middleware.

Let's kick off our custom middleware creation with maybe the most classic middleware example around: logging.

NOTE **If you're coding along, pause here to make sure you commit all changes up to this point. Future chapters will require rolling back features created in this chapter.**

5.3 Example: logging middleware

The goal: for every action dispatch, log the action being dispatched (including type and payload), and the new state of the store after the action is processed.

Because we need to run code for every action that's dispatched, logging fits the use case for middleware perfectly. It's also simple to implement, because it doesn't affect the normal flow of control within the app. We're not modifying the action or changing the outcome of a dispatch in any way. We want to hook into only the action lifecycle and log certain details about the action being dispatched and how it affects the state of the store.

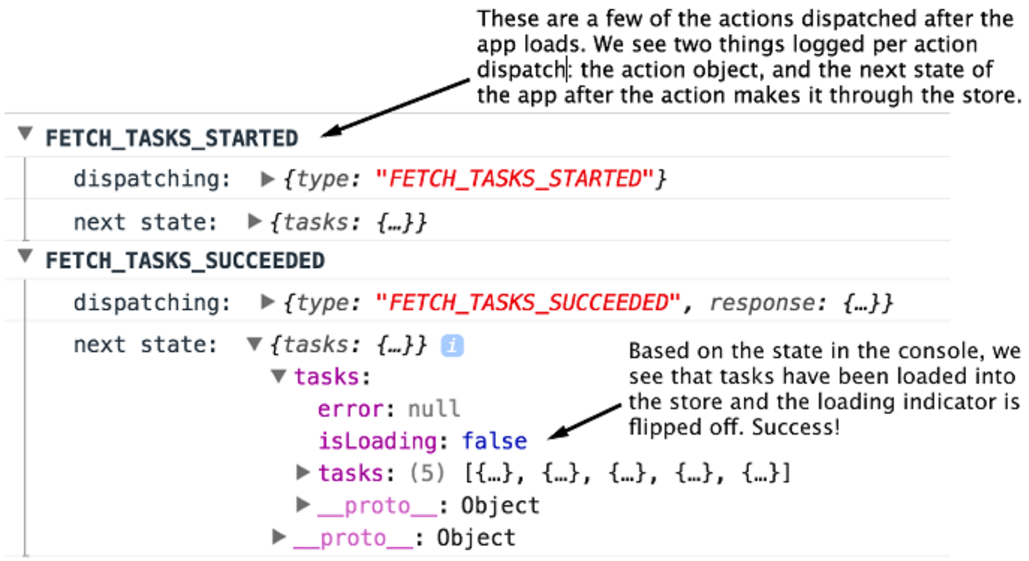
It's also a bonus that logging fits Redux perfectly. Actions were invented in part to provide a trail of the events and data flying around our applications. Because events are modeled as objects with descriptive names and any data they might need, it's trivially easy for us to log an action's type and payload to get a quick sense of exactly what's happening at any given moment. Without a system like Redux, where updates must pass through a central hub, it would be much more difficult to log state changes with the same kind of effort.

5.3.1 Creating the logger middleware

Given that this is the first custom middleware we'll be adding, let's take a high-level view at what it takes to go from zero to a having a functioning logging middleware as part of our application:

- Create a new file for the logger in a `src/middleware/` directory
- Write the middleware
- Import the middleware into `index.js` where we create the store
- Add the middleware using Redux's `applyMiddleware` function

Not so bad, right? For the middleware itself, we want to log two things: the type of the action being dispatched, and the state of the store after it processes the action. Figure 5.2 shows the kind of output you'll see in the console after you've created the middleware and registered it with the store.


These are a few of the actions dispatched after the app loads. We see two things logged per action dispatch: the action object, and the next state of the app after the action makes it through the store.
Based on the state in the console, we see that tasks have been loaded into the store and the loading indicator is flipped off. Success!

```
▼ FETCH_TASKS_STARTED
    dispatching:  ▶{type: "FETCH_TASKS_STARTED"}
    next state:  ▶{tasks: {…}}
▼ FETCH_TASKS_SUCCEEDED
    dispatching:  ▶{type: "FETCH_TASKS_SUCCEEDED", response: {…}}
    next state:  ▼{tasks: {…}}
                   ▼tasks:
                      error: null
                      isLoading: false
                    ▶tasks: (5) [{…}, {…}, {…}, {…}, {…}]
                    ▶__proto__: Object
                  ▶__proto__: Object
```

Figure 5.2 Example console output after adding the logger middleware.

Start by creating a new directory `src/middleware/`, and a new file named `logger.js`. Inside the file, add the code from listing 5.1. Here we use all three of the arguments provided to our middleware, each to perform a different task:

- We use `action` to log the type of the action being dispatched
- We use `store` to log the state of the store after applying the action with `store.getState`
- And finally, we use `next` to pass the action to the next middleware in the chain

Listing 5.1 Creating the logger middleware – src/middleware/logger.js

```
const logger = store => next => action => {  ❶
  console.group(action.type);  ❷
  console.log('dispatching: ', action);
  const result = next(action);  ❸
  console.log('next state: ', store.getState());  ❹
  console.groupEnd(action.type);
  return result;
};

export default logger;
```

❶ Create the middleware using the correct function signature
❷ Use console.group to style console output
❸ Use next to ensure the action gets passed to the reducers and the next state is calculated
❹ Log the state of the store after the action has been applied

By now you know what action objects are and that they have a required `type` property, and you know how to retrieve the current state from the store with `store.getState`, so the `next` function is likely the most foreign concept in play here.

`next`, provided to us by Redux, is a way for us to signify when this middleware has completed its work and it's time to move on to the next middleware in the chain (if there is one). It's effectively a wrapped version of `dispatch`, so it has an identical API. Ensure you're always passing in the `action` argument whenever you call `next` within a middleware.

One curiosity here is that the middleware doesn't just end after we use `next(action)`. We're free to carry on and reference the Redux state after the action passes through the logger middleware.

5.3.2 Using applyMiddleware to register the middleware

At this point we have a perfectly valid logger middleware, but it's not particularly useful on its own. To use it in Parsnip, we'll have to add the middleware to our store instance. If you've worked through chapter 4, then this should be familiar! It's the same process we went through to add the `redux-thunk` middleware when we introduced async actions.

Open `index.js`, import the logger, and add it to the argument list for `applyMiddleware` alongside `thunk`. Whenever you want to add a new middleware to your applications, you'll follow this process. Note that we add the logger middleware last, and they run in the order that you pass them to `applyMiddleware`. It's not critical in this case, that the middleware be in any particular order, but you should be aware of the significance of their order.

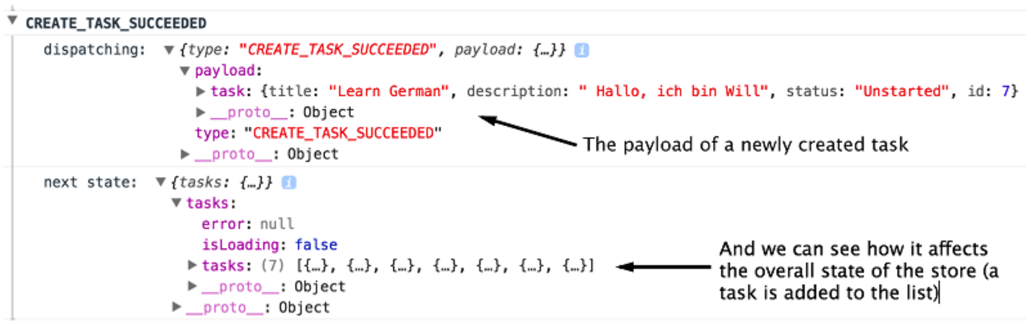
Listing 5.2 Adding the logger middleware to the store – src/index.js

```
...
   import logger from './middleware/logger';              ❶
   ...

   const store = createStore(
     rootReducer,
     composeWithDevTools(applyMiddleware(thunk, logger)), ❷
   );

   ...
```

❶ Import the newly created middleware
❷ Register the logger middleware with the store

The middleware system is designed to be flexible and composable. As long as each middleware calls `next` to move to the next middleware in the chain, everything will work correctly.

After this last step of registering the middleware when the store is created, Parsnip has its first fully-functioning custom middleware. Take a look at the app and open the browser console, where you can see the fruits of your labor. Try creating a new task or editing an existing task. You should see output similar to figure 5.3.



```
▼ CREATE_TASK_SUCCEEDED
  dispatching:  ▼{type: "CREATE_TASK_SUCCEEDED", payload: {…}}
                  ▼payload:
                    ▶task: {title: "Learn German", description: " Hallo, ich bin Will", status: "Unstarted", id: 7}
                    ▶__proto__: Object
                   type: "CREATE_TASK_SUCCEEDED"
                  ▶__proto__: Object
  next state:  ▼{tasks: {…}}
                 ▼tasks:
                    error: null
                    isLoading: false
                   ▶tasks: (7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
                   ▶__proto__: Object
                 ▶__proto__: Object
```

The payload of a newly created task
And we can see how it affects the overall state of the store (a task is added to the list)

Figure 5.3 Another example of console output from the logger middleware.

You might be asking "isn't this just a less powerful version of the Redux DevTools?", and you'd be correct! The dev tools are implemented using very similar concepts as the middleware we've just created. We decided to start with middleware for logging because of how cleanly it demonstrates one of the core use cases for Redux middleware. In your own apps, you may choose to use a middleware like this if you want to keep things simple, but most likely you'll get the most bang for your buck with an established tool like the DevTools.

Parsnip now has its first custom middleware, but we've only scratched the surface. Let's explore another common use case for middleware: sending analytics events. The analytics middleware will be similar to the logger, but we'll introduce a new concept along the way: the `meta` property, which allows us to attach additional data to an action object. Then, in the middleware we can watch for the `meta` property on incoming actions and do some work if the right conditions are satisfied.

5.4 Example: analytics middleware

Knowing what your users are doing is always preferable to not knowing, so in any real-world application, it's a good idea to have proper event tracking. In Parsnip, we can add tracking to most of the actions currently being dispatched. After all, actions are just objects that describe an event, so we can hook into an action dispatch and record an analytics event whenever users view a page, create a task, or edit a task. We won't go as far as actually setting up a third-party analytics service like Segment or Google Analytics, but we'll implement a Redux middleware that provides a convenient API for implementers of Parsnip to send new events. Analytics is a good use case for middleware for a few reasons:

- Like the other middleware we've worked with so far, we need to run analytics code across many different actions
- It allows us to abstract the specifics of event tracking, like which service is being used (for example, Segment, Keen) and any configuration.

For something like analytics, we're using middleware to encapsulate implementation details and provide a developer-friendly way to send analytics events. Other Parsnip developers won't need to know the specifics of how to send an event, they only need to know the high-level interface available to them.

5.4.1 The meta property

So far, we've only dealt with two top-level properties as part of an action: `type`, a string which declares which action is being dispatched, and `payload`, any data that the action needs to be completed. There's a third action property that's been gaining popularity within the community, the `meta` property. `meta` is designed to capture any data relevant to an action that doesn't fit within either `type` or `payload`. For our use case, we'll use it to send along analytics data, specifically an event name and some metadata.

In the yet-to-be-created analytics middleware, we'll watch for any action that has the relevant `meta` property. Every analytics event has two components: an event name, and any metadata that the event might require. Let's name this event create_task, and pass along the ID of the task being created.

Fire an analytics event whenever the action is dispatched by heading to `src/actions/index.js` and updating the `createTaskSucceeded` action creator to match listing 5.3.

Listing 5.3 Adding the meta property – src/actions/index.js

```
function createTaskSucceeded(task) {
  return {
    type: 'CREATE_TASK_SUCCEEDED',
    payload: {
      task,
    },
    meta: {           ❶
      analytics: {   ❷
        event: 'create_task',
        data: {
          id: task.id,
        },
      },
    },
  };
}
```

❶ Add the meta property at the same level as type and payload
❷ Group analytics-related data under a namespace key

Indirection in the code is one of the potential downsides or costs of using middleware, but it isn't too much of an issue here. As we'll see in a moment, the action creator still gets to be explicit. Because we will be using the meta property directly on the action object, anyone reading the action creator will know that analytics will be captured whenever the action is dispatched. However, they won't need to know specifics about how to send events data. As users of the middleware, we can simply pass the right data with the right structure, and let the middleware take care of the rest.

Now you have a sense of how we'll pass analytics data along with an action, but that's only one piece of the puzzle. We're using the `meta` property in the action creator, but we also need the middleware to take care of watching for actions with said `meta` property. In the next section we'll create the middleware, and we'll update a few of our actions to send analytics events by adding `meta`.

5.4.2 Adding analytics middleware

Now on to the middleware itself! Our goal is to create a middleware that, when an applicable action is being dispatched, will take care of everything related to recording an event. The action is responsible for passing any event data; the middleware is responsible for encoding the details of how to send that event. For this example, we'll mock out an analytics API. In a production app however, you'd likely be using a third-party service, so the middleware is where you'd include any library code.

Here's how the flow of control in the middleware might look for dispatching an action with the `meta` and `analytics` properties:

- For each incoming action, check if the meta and analytics properties exist.
- If not, call `next(action)` to move on to the next middleware.
- If so, get the event name and data, and send it using a fictional analytics service.
- And finally, when that's complete, call `next(action)`.

We already have the `CREATE_TASK_SUCCEEDED` action sending the `meta`/`analytics` properties, now we need to take a stab at implementing the middleware to satisfy the above criteria. Create a new file in `src/middleware/` called `analytics.js`, and add the code from listing 5.4.

Here, we introduce something that's a common practice you'll see with middleware. Instead of doing some work for every action dispatch like we did with the logger, we instead check for certain conditions before letting the middleware take over. In this case, if we see that the action has the `meta` and `analytics` properties, we know that the action has requested to track an event. In all other cases, we'll call `next(action)` right away without doing any further processing.

Listing 5.4 Implementing the analytics middleware – src/middleware/index.js

```
const analytics = store => next => action => {
  if (!action || !action.meta || !action.meta.analytics) {   ❶
    return next(action);          ❷
  }

  const { event, data } = action.meta.analytics;             ❷

  fakeAnalyticsApi(event, data)   ❸
    .then(resp => {
      console.log('Recorded: ', event, data);                ❹
    })
    .catch(err => {
      console.error(               ❺
        'An error occurred while sending analytics: ',
        err.toSting(),
      );
    });

  return next(action);             ❻
};

function fakeAnalyticsApi(eventName, data) {
  return new Promise((resolve, reject) => {
    resolve('Success!');
  });
}

export default analytics;
```

❶ Check if the action wants to use the analytics middleware
❷ If the right action properties aren't found, move on to the next middleware
❸ Use destructuring to get the event name and any associated metadata
❹ Log the event being recorded successfully
❺ Log an error if necessary
❻ Move on to the next middleware

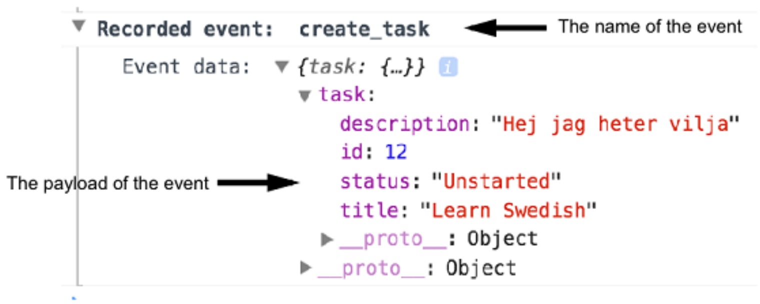
As with the logger middleware, we'll also need to register the middleware with the store in `src/index.js`.

Listing 5.5 Applying the analytics middleware – src/index.js

```
...
   import analytics from './middleware/analytics';
   ...

   const store = createStore(
     rootReducer,
     composeWithDevTools(applyMiddleware(thunk, logger, analytics)),   ❶
   );
   ...
```

❶ Register the analytics middleware by passing it to applyMiddleware

Head to the app, make sure your browser console is open, and try creating a new task. You should see output similar to figure 5.4, indicating that our analytics middleware is working properly. We're looking specifically for the event name, `create_task`, and the task object that we passed to the analytics middleware.



```
▼ Recorded event:  create_task
    Event data:  ▼ {task: {…}}
                   ▼ task:
                       description: "Hej jag heter vilja"
                       id: 12
                       status: "Unstarted"
                       title: "Learn Swedish"
                     ▶ __proto__: Object
                   ▶ __proto__: Object
```

Figure 5.4 Example output for the create_task analytics event.

With logging, we did some work in the middleware for every action, but not every middleware you write will want to respond in such a way. Here we're doing something slightly different, but also extremely common: checking if the action being dispatched is one we care about. If it's not relevant to the middleware, we'll bail out and send the action on to the next middleware. If it's an action we care about, we'll take care of sending analytics before we move on.

This pattern of using a meta property to allow actions to indicate to a middleware that it should do some work is great for a few reasons:

- it allows us to be explicit at the action level. Because the analytics data is baked directly in to the action object, it's very clear that the action wants to do something analytics related.
- It lets the middleware stay generic. Instead of having the middleware watch for specific actions, letting actions specify they want some additional work done via the `meta` property allows the middleware itself to remain relatively static.
- We now have an extremely friendly API for developers. Sending along an analytics object with an action is easier than having to import and use any analytics code directly.

Think of it as the best of both worlds. We get to abstract analytics functionality into the middleware, meaning we don't have to duplicate it around the app, but we also get to define actions in an explicit way.

INFO Though we won't do it here, the middleware itself is also a great way to encapsulate any data that might apply to all analytics events. Because you have access to the state of the store, it's easy to get things like the id of a logged-in user, or an identifier for the current build of your application.

If you want to add tracking to additional actions, follow a similar pattern to what we did with the `CREATE_TASK_SUCCESS` action. Using the `meta` property, specify an event name, any data the event might need, and let the middleware take care of the rest!

Now that you have experience creating two custom middleware, it's a good time for brief conceptual interlude. Let's pause here to explore potential middleware pitfalls.

5.4.3 Interlude: when and when not to use middleware

The real benefit of middleware is being able to centralize certain tasks that need to be applied across many actions. Again, let's use logging as an example. Say our goal is to log the `type` and `payload` of every action being dispatched. One way to accomplish this is to add logging statements at the point of every call to `store.dispatch`. Sure, we're logging action dispatches, which was our original goal, but this solution should make you cringe for at least one reason: it's not particularly scalable. Now we'll have to add logging for every action we create.

Middleware allows us to bypass this entirely by defining the logic for logging actions in a single place. All new actions will get logging for free, without any developer intervention.

So, when is it a good idea to use middleware? For us, there are two golden rules:

- Use middleware if you need to write code that applies across many, if not all actions in the application. Logging is maybe *the* most classic example.
- But, only if the indirection caused by the middleware is not overly damaging to the readability and understandability of the code.

There's a lot of nuance in that last bit, and it's a good segue into an equally important question: when should we NOT use middleware?

Redux middleware in the right scenarios is incredibly useful and powerful, but like many things in life, it's possible to have too much of a good thing. The main tradeoff with middleware is indirection. This usually isn't an issue with big cross-cutting concerns like logging, but you have to be wary when using middleware for tasks that affect the overall flow of control within your apps. For example, we'll build an API middleware later in the chapter which helps centralize tasks common to all async actions that require an AJAX request. It's a powerful abstraction, but because it directly impacts the flow of data, it's an added layer of complexity that's practically impossible for developers working in that area of the code to ignore.

Usage in cases like these will, as always, depend on the current situation and factors like the size of your team and how much code you're dealing with. Redux is here to help us take back control of the flow of data through our applications, and responsible middleware usage is a big part of that.

5.4.4 Case study: how not to use middleware

Logging and analytics are great examples for when you *should* reach for middleware, but it's not always that light and breezy of a decision. It's worth covering the potential for misuse with middleware, so we'll use routing as a case study. The problem we're about to outline approximates something we dealt with in a real-world application, and highlights one of many hard lessons around best practices with middleware learned along the way.

Once upon a time, we had an application that needed a place for core routing code. Specifically, the goal was to redirect users to a dashboard after logging in. We decided to use middleware, but wrote it in a very non-generic way. Instead of allowing actions to indicate they needed to perform a redirect, we watched for specific actions like `LOGIN_SUCCESS` directly in the middleware.

From a readability perspective, we eventually lost the mental link between actions that triggered subsequent dispatches. The routing middleware became the de facto place for all routing, and the code blew up over time. In retrospect, it would have been better for us to use a more generic approach, something more along the lines of the `meta` property we used for analytics.

Again, the upside of the `meta` strategy is that it lets us keep actions explicit. Reading the action conveys everything you need to know that might affect the flow of control. The downside is that action creators become slightly larger, and take on an additional responsibility. Ultimately, like everything in software, the best solution depends on your realities.

Let's look at some of the potential solutions to the original problem statement: a successful login should redirect the user to the `/dashboard` route. Listing 5.6 shows how we might do it using thunks and async action creators, which we learned about in chapter 4. Note that this code won't be part of Parsnip.

Listing 5.6 Redirecting after login

```
export function login(params) {
  return dispatch => {
    api.login(params).then(resp => { #//A
      dispatch(loginSucceeded(resp.data));   ❷

      dispatch(navigate('/dashboard'));      ❸
    });
  };
}
```

❶ Make the call to login
❷ Dispatch an action indicating login was successful
❸ Perform the redirect

If you needed to interact with this `login` action creator, everything you need to know is contained within it. It's more imperative; the code reads like a list of step-by-step instructions, but `login` is transparent about its responsibilities.

On the other hand, you can add specific routing logic in a middleware like we did, shown in listing 5.7. Instead of dispatching the `navigate` action directly within the `login` action, we'll move that logic to a routing middleware. Middleware by default has a chance to inspect and respond to all actions, so in this case we'll wait until we see `LOGIN_SUCCEEDED` action to redirect the user.

Listing 5.7 Routing logic in a middleware

```
function login(params) {
  return dispatch => {
    api.login(params).then(resp => {
      dispatch(createTaskSucceeded(resp.data));
    });
  };
}

const notifications = store => next => action => {
  if (action.type === 'LOGIN_SUCCEEDED') {   ❶
    dispatch(navigate('/dashboard'));        ❷
  }
};
```

❶ **Watch for the LOGIN_SUCCEEDED action**
❷ **Check if the action being dispatched is the one we care about**

This code seems harmless at first, and maybe it is, assuming the routing middleware doesn't grow too much. But it has one critical flaw – indirection. In our experience, it's usually a mistake to use middleware in this way.

You may be familiar with the decades-old concept of "the principle of least surprise." Liberally paraphrased, if users are consistently surprised by some functionality, it might be time to re-think that experience. This user experience guideline may be extended to developers as well: if an implementation is surprising, consider an alternative. Future implementers of `login` need to somehow gain the unintuitive context of the middleware. If they're new to the project or Redux, there's a good chance they'll miss it at first, and be confused as to what exactly is triggering the redirection.

Let's look at this in a more visual way. Figure 5.5 illustrates the two different approaches for handling the redirect following a successful login.

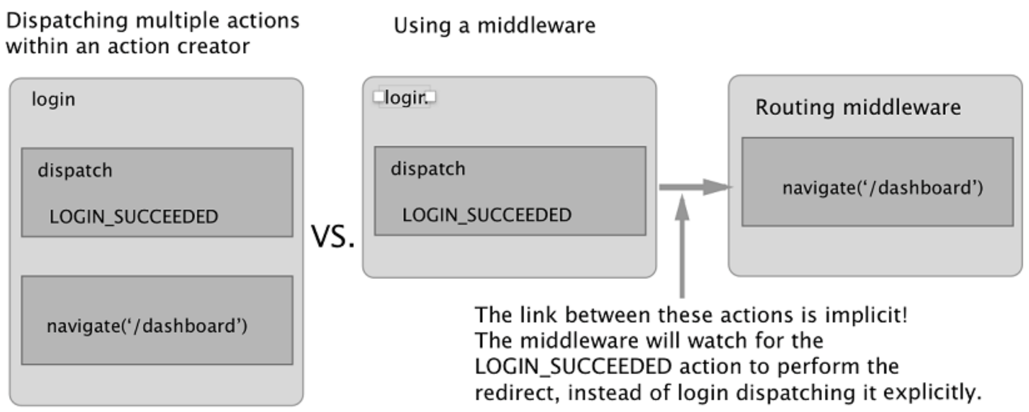

Dispatching multiple actions within an action creator
login
dispatch
LOGIN_SUCCEEDED
navigate('/dashboard')
VS.
Using a middleware
login
dispatch
LOGIN_SUCCEEDED
Routing middleware
navigate('/dashboard')
The link between these actions is implicit! The middleware will watch for the LOGIN_SUCCEEDED action to perform the redirect, instead of login dispatching it explicitly.

Figure 5.5 Modeling different strategies for routing as part of an async action.

The key takeaway here is the difference between explicit vs. implicit approaches. With the action creator strategy on the left, we're explicit about the additional work taking place in the `login` example.

This is a small example, but picture middleware like this handling 5, 10, or 20 actions. Sure, much of the code around routing will be bundled in one place, but the flow of control for any individual action dispatch becomes more difficult to track down. Middleware can help you reduce duplication and centralize logic, but it comes at a cost, and it's up to you to use your best judgement. Middleware are abstractions and their primary purpose is to aid the development process. Like all abstractions, there is a point where they can go too far and negatively impact the overall design of your application.

Let's implement one more middleware, this time for API calls to the server, which will have the highest degree of difficulty.

5.5 API middleware

We now know that middleware is meant to abstract logic that's common to many actions. Looking at Parsnip's existing functionality, is there any logic that we can abstract? Think about the actions in the app that make API calls to the server. So far, we have `fetchTasks`, `createTask`, and `editTask`. What do they all have in common? They all seem to:

- Dispatch an action indicating that a request has started
- Make an AJAX request
- If the request succeeds, dispatch an action with the response body
- If the request fails, dispatch an action with the error message

If you recall from chapter 4, these steps relate to the three key moments in time for any server API call. We model each of these events with a standard set of actions types. Using task creation as an example, we dispatch three actions:

- `CREATE_TASK_STARTED`
- `CREATE_TASK_SUCCEEDED`
- `CREATE_TASK_FAILED`

For any future action that you implement that requires an AJAX request, you'll have to create these three corresponding action types. There's nothing inherently wrong with this strategy. It has its own benefits, mainly that it's a very explicit way to model the interactions in your application. But, it's heavy on the boilerplate. We're doing the same type of work, just with different endpoints and different data. We can attempt to centralize much of the logic around sending requests by creating a new middleware to take care of the tasks common amongst all of these async actions.

In our opinion, things like logging and analytics are no-brainers for middleware. They're generic, run across many actions, and don't interrupt the normal flow of control. The API middleware we're about to implement is slightly different. It directly affects how you interact with one of the core pieces of functionality in any application: AJAX requests. Whether you think adding this middleware is worth it in your own projects is up to you. Ultimately, the benefits are centralization of logic, at the cost of added complexity. Parsnip is all about exploring and experimenting with new technologies, so let's go for it and see how an API middleware like this might come together.

NOTE **This type of API middleware is popular both in production and as a teaching example, because it's a great example of extracting repeated functionality. We took plenty of inspiration from two places: the "Real World" example that's part of the official Redux docs (https://github.com/reactjs/redux/blob/master/examples/real-world), as well as a popular open-source variant, (https://github.com/agraboso/redux-api-middleware/issues).**

5.5.1 The desired API

What should the action creators look like? For this middleware, we'll require all actions with API calls to have three things:

- A `CALL_API` field defined and exported by the middleware
- A `types` property, an array of the three action types for the request starting, completing successfully, and failing
- An `endpoint` property, to specify the relative URL of the resource we want to request

We already have an existing implementation of the `fetchTasks` action, but for this middleware we'll replace it completely with a new implementation. The good news is that we'll have to make minimal updates elsewhere in the app. The `App` component, which dispatches the `fetchTasks` action, will notably require zero changes. Under the hood, the implementation of how the app fetches and stores tasks has changed, but the view layer (React components) is safely isolated from any of this work.

Using the code in listing 5.7, do the following:

- Import a CALL_API action from the yet-to-be-created middleware. This will be how `fetchTasks` will indicate that it wants to use the API middleware.
- Define and export constants for each action related to `fetchTasks`: FETCH_TASKS_STARTED, FETCH_TASKS_SUCCEEDED, and FETCH_TASKS_FAILURE.
- From `fetchTasks`, return an object that uses the CALL_API action, passes the three action types, and finally passes the /tasks server endpoint.

Implementing the above will mean we'll have the data we need when we get to the middleware itself in the next section. As we'll see in a moment, the order in which we add the three action constants within the `types` array matters. The middleware will assume that the first action is request start, the second is request success, and the third is request failure.

TIP If the CALL_API syntax is new to you, the square brackets were introduced in ES6 and evaluate the variable within them to dynamically produce a key.

Listing 5.8 Updating fetchTasks – src/actions/index.js

```
...
import { CALL_API } from '../middleware/api';  ❶

export const FETCH_TASKS_STARTED = 'FETCH_TASKS_STARTED';      ❷
export const FETCH_TASKS_SUCCEEDED = 'FETCH_TASKS_SUCCEEDED';  ❷
export const FETCH_TASKS_FAILURE = 'FETCH_TASKS_FAILURE';      ❷

export function fetchTasks() {
  return {
    [CALL_API]: {          ❸
      types: [FETCH_TASKS_STARTED, FETCH_TASKS_SUCCEEDED,  FETCH_TASKS_FAILURE],  ❸
      endpoint: '/tasks',  ❸
    },
  };
}

// function fetchTasksSucceeded(tasks) {       ❹
//   return {
//     type: 'FETCH_TASKS_SUCCEEDED',
//     payload: {
//       tasks,
//     },
//   };
// }
//
// function fetchTasksFailed(error) {
//   return {
//     type: 'FETCH_TASKS_FAILED',
//     payload: {
//       error,
//     },
//   };
// }
//
// function fetchTasksStarted() {
//   return {
//     type: 'FETCH_TASKS_STARTED',
//   };
// }
//
// export function fetchTasks() {
//   return dispatch => {
//     dispatch(fetchTasksStarted());
//
//     api
//       .fetchTasks()
//       .then(resp => {
//         dispatch(fetchTasksSucceeded(resp.data));
//       })
//       .catch(err => {
//         dispatch(fetchTasksFailed(err.message));
//       });
//   };
// }

...
```

1. **Import the CALL_API constant**
2. **Define three constants for each action dispatched within fetchTasks**
3. **From fetchTasks, return the data the middleware will need**
4. **Comment (or delete) the existing implementation of fetchTasks**

We just removed a whole lot of functionality from the `fetchTasks` action, and that's the whole point! The next step is to move request logic into a centralized place. In the next section, we'll create the API middleware that knows how to deal with an action like the one our new version of `fetchTasks` returns.

5.5.2 Outlining the API middleware

Because this middleware is more complex than the logger and analytics middlewares, we'll create it in several smaller steps. Create a new file in the `src/middleware/` directory named `api.js`. Using listing 5.9, start by creating the required middleware boilerplate: defining and exporting the main middleware function. Next, define the `CALL_API` action, and check if the current action includes the `CALL_API` type. If it doesn't, pass the `action` to `next` to move on the next middleware.

Note the line in the listing where we check if `callApi` is undefined. We followed a similar process with the analytics middleware, where we checked for the existence of a `meta` property. In both cases, if the current action didn't satisfy the criteria for the middleware, we immediately called `next(action)` and moved on. This is a common pattern known as a *guard clause*. We're huge fans of guard clauses for readability purposes. They allow us to define exceptional cases for a function up front, freeing up the function body to be a clean, un-indented block. Once we get past a guard clause, we can assume any data that the middleware might need will be available.

Listing 5.9 Sketching out the API middleware – src/middleware/api.js

```
export const CALL_API = 'CALL_API';        ❶

const apiMiddleware = store => next => action => {
  const callApi = action[CALL_API];        ❷
  if (typeof callApi === 'undefined') {    ❸
    return next(action);                   ❸
  }
}

export default apiMiddleware;
```

❶ **Define the middleware-specific action constant**
❷ **Get the object with the types and endpoint properties (if it exists)**
❸ **If It's not an action meant for this middleware, continue without any further work**

Because we now have a totally functional, albeit not very useful, middleware, take this opportunity to register it with the store so we can use it within Parsnip. Head to `src/index.js`, import the middleware, and pass it to the `applyMiddleware` function.

Listing 5.10 Registering the API middleware – src/index.js

```
...
import apiMiddleware from './middleware/api';
...
const store = createStore(
  rootReducer,
  composeWithDevTools(applyMiddleware(thunk, apiMiddleware, logger, analytics)),
);
```

This time, the order in which we apply the middleware does matter. Because the API middleware requires a custom action shape, we want to include it first. If the logger or analytics middleware came before the API middleware, they wouldn't know how to handle an action without a `type` property and would throw an exception.

Now we can get to the meat of the middleware. We need to dispatch the first of our three actions, which will indicate that a request has started. Add the code from listing 5.11 to `src/middleware/api.js`.

Listing 5.11 Dispatching the request started action – src/middleware/api.js

```
...

const apiMiddleware = store => next => action => {
  const callApi = action[CALL_API];
  if (typeof callApi === 'undefined') {
    return next(action);
  }

  const [requestStartedType, successType, failureType] = callApi.types;   ❶

  next({ type: requestStartedType });   ❷
}

...
```

❶ **Using array destructing to create variables with each action type**
❷ **Dispatch the action which indicates the request is in progress**

Because `next` will ultimately dispatch an action to the store, we pass it an action object just like we would if we were using `store.dispatch`. The result is identical to the old strategy of dispatching `FETCH_TASKS_STARTED` directly within the `fetchTasks` action. The reducer will update the correct state in response to the action, and the app will know to render a loading indicator.

The middleware uses array destructuring to create variables for each action type, which was why it was important to add the action types in the right order when we implemented the new fetchTasks action.

Next, let's add a function for making the AJAX call. We already have an API client that lives at `src/api/index.js`, but for our purposes here we need a new, more generic function that accepts an endpoint as an argument. Update the code in `src/middleware/api.js` to match listing 5.12.

Listing 5.12 The makeCall function – src/middleware/api.js

```
import axios from 'axios';

const API_BASE_URL = 'http://localhost:3001';   ❶

export const CALL_API = 'CALL_API';

function makeCall(endpoint) {
  const url = `${API_BASE_URL}${endpoint}`;     ❷

  return axios
    .get(url)
    .then(resp => {
      return resp;   ❸
    })
    .catch(err => {
      return err;   ❸
    });
}

...
```

❶ **Define the base url for the API**
❷ **Use the given endpoint to construct a final request URL**
❸ **Return the response and error respectively from each promise handler**

`makeCall` is generic enough to work with our middleware. We'll pass it an endpoint defined by the action being dispatched, and then `makeCall` will return either a response or an error depending on the result of the request.

5.5.3 Making the AJAX call

Next let's go ahead and use the `makeCall` function we just created. We've already dispatched the action to indicate the request has started. Now, we'll make the API call and dispatch either the success action or the failure action depending on the result.

Listing 5.13 Making the AJAX call – src/middleware/api.js

```
...
const apiMiddleware = store => next => action => {
  const callApi = action[CALL_API];
  if (typeof callApi === 'undefined') {
    return next(action);
  }

  const [requestStartedType, successType, failureType] = callApi.types;

  next({ type: requestStartedType });

  return makeCall(callApi.endpoint).then(   ❶
    response =>
      next({   ❷
        type: successType,
        response,
      }),
    error =>
      next({   ❸
        type: failureType,
        error: error.message,
      }),
  );
};

export default apiMiddleware;
```

❶ **Pass the endpoint specified by the current action to makeCall**
❷ **If the request succeeds, dispatch the success action type with the response**
❸ **If it fails, dispatch the failure type with an error message**

And thus we've reached API nirvana. In this middleware, we've managed to create a centralized place that implements a few of the tasks that are common across all AJAX requests we'll need to make in the app. The main benefit here is that we'll be able to drastically reduce any future boilerplate that might come along if we add additional async actions that require a server request. Instead of creating three new action types and dispatching them all manually, we can use the API middleware to do the heavy lifting for us.

5.5.4 Updating the reducer

We're finished with all the middleware related work, but there's one last step we need to take. Ideally, we'd be able to update the implementation of `fetchTasks` without updating other components, like the reducer, but to keep the middleware generic, we have to make small a concession and update the reducer to handle a slightly less friendly action payload. Update the handler for the `FETCH_TASKS_SUCCEEDED` action in `src/reducers/index.js` to use the payload defined by the API middleware.

Listing 5.14 Updating the tasks reducer – src/reducers/index.js

```
const initialState = {
  tasks: [],
  isLoading: false,
  error: null,
};

export default function tasks(state = initialState, action) {
  switch (action.type) {
    ...
    case 'FETCH_TASKS_SUCCEEDED': {
      return {
        ...state,
        tasks: action.response.data,    ❶
        isLoading: false,
      };
    }

    ...
    default: {
      return state;
    }
  }
}
```

❶ **Use the new action shape to update tasks state**

This is a change we'd rather not make, but it's not a disaster. New tasks being available in the reducer at `action.payload.tasks` was more descriptive, but it's a small price to pay in the larger picture. Libraries like normalizr can allow the API middleware to dispatch action payloads that are more specific, and we'll cover that process in a subsequent chapter.

5.5.5 Wrapping up API middleware

And there you have it: a powerful API middleware that helps centralize the common themes around sending an AJAX request with Redux. But remember, abstractions all have a cost, and the cost here is code complexity. Consider the tradeoffs, and do whatever makes the most sense given your project realities.

5.6 Exercise

Since we've built this nice new API middleware, it would only make sense to update some of the other async actions to use it. See if you can migrate `createTask` similarly to how we migrated `fetchTasks` in the previous section. `createTask` currently dispatches the three typical request actions, request start, success, and failure.

It's not quite as simple as only implementing a new `createTask` that uses the API middleware. You'll also have to update the middleware itself to support a POST request in addition to a GET request.

5.7 Solution

Here's how we might break down this problem into manageable chunks:

- Update the API middleware to accept a request method and a request body, which it will use when makes the AJAX request
- Implement a new version of `createTask` that uses the `CALL_API` action constant, and passes four arguments to the API middleware: an array of the three request-related actions, the endpoint, the request method, and any POST data
- Update the reducer to handle the new action payload for CREATE_TASK_SUCCEEDED as supplied by the API middleware

First, update the middleware to handle POST requests.

Listing 5.15 Updating the API middleware – src/middlewares/api.js

```
function makeCall({ endpoint, method = 'GET', body }) {   ❶
  const url = `${API_BASE_URL}${endpoint}`;

  const params = {
    method: method,
    url,
    data: body,
    headers: {
      'Content-Type': 'application/json',
    },
  };

  return axios(params).then(resp => resp).catch(err => err);
}

...

const apiMiddleware = store => next => action => {
  ...

  return makeCall({
    method: callApi.method,   ❷
    body: callApi.body,       ❷
    endpoint: callApi.endpoint,
  }).then(
    response =>
      next({
        type: successType,
        response,
      }),
    error =>
      next({
        type: failureType,
        error: error.message,
      }),
  );
};
```

1 Update makeCall to accept a method/request body
2 Pass the new arguments provided by the action

It required a small effort, but this is a big win for us. We only had to add a few lines, but in the process, we made the middleware much more flexible for users of our code. Next, take care of updating `createTask` to use the `CALL_API` action, making sure to also provide a request method and body. Similar to `fetchTasks`, we're able to remove a ton of boilerplate here in favor of a much more declarative strategy.

Listing 5.16 Implementing the new createTasks – src/actions/index.js

```
export const CREATE_TASK_STARTED = 'CREATE_TASK_STARTED';       1
export const CREATE_TASK_SUCCEEDED = 'CREATE_TASK_SUCCEEDED';   1
export const CREATE_TASK_FAILED = 'CREATE_TASK_FAILED';         1

export function createTask({ title, description, status = 'Unstarted' }) {
  return {
    [CALL_API]: {
      types: [CREATE_TASK_STARTED, CREATE_TASK_SUCCEEDED, CREATE_TASK_FAILED],   2
      endpoint: '/tasks',   3
      method: 'POST',       3
      body: {               3
        title,
        description,
        status,
      },
    },
  };
}
```

1 Create new action constants for the request actions
2 Ensure that they are passed in the correct order
3 Pass the required arguments, including the new method/body properties

We're almost there, but we have one last small step. Similar to `fetchTasks`, we need to update the reducer to accept a new action shape for `CREATE_TASKS_SUCCEEDED`. Because we haven't added any logic to have the API middleware dispatch actions with custom payloads, the best we can do is pass the entire response object into the reducer. Head to `src/reducers/index.js` and make the final change according to listing 5.17.

Listing 5.17 Updating the tasks reducer – src/reducers/index.js

```
...
  case 'CREATE_TASK_SUCCEEDED': {
    return {
      ...state,
      tasks: state.tasks.concat(action.response.data),
    };
  }
...
```

Very nice! If you're feeling industrious, update the `editTask` action to also use the API middleware. Overall, what style do you prefer? Would you rather have each action creator (`fetchTasks`, `createTasks`) explicitly dispatch multiple actions using redux-thunk? Or do you prefer the power of the API middleware? Redux isn't a large, opinionated framework, so there's always more than one way to peel the orange.

5.8 Summary

Middleware is fundamental to Redux, and one of its most powerful features. But with great power, comes great responsibility. Middleware is a great way to centralize code and reduce duplication, and sometimes to create friendlier, more powerful APIs for yourself and for your collaborators.

In this chapter, you learned the following:

- Redux middleware is code that sits between an action being dispatched and the store processing the action
- It's best used for generic tasks like logging, that need to be applied to many, if not all actions
- It can produce powerful abstractions like the API middleware, but often come at the cost of complexity and indirection

In the next chapter, we'll explore another popular abstraction for async actions. Did you heed the warning in this chapter to commit your work before implementing all the middleware? Before beginning the next chapter, rollback your code to that commit. We'll need a clean slate before introducing Redux Sagas.

6
Handling complex side-effects

This chapter covers

- **Another look at redux-thunk**
- **An introduction to generators**
- **Managing complex asynchronous operations with sagas**

Eventually, you're going to want to handle more complex series of events in response to a user interaction. Using what you've learned so far, how would you handle logging in a user? The requirements will vary from app to app, of course, but let's consider what could be involved. Our application may need to verify login credentials, issue an authorization token, fetch user data, handle retries upon failure, and redirect upon success. What tools are at your disposal?

So far, we've explored thunks, via the `redux-thunk` package, as a way to handle side-effects and asynchronous operations. Thunks have a friendly learning curve and are powerful enough to handle just about any use case you can throw at them. However, they aren't the only game in town. In this chapter, we'll revisit thunks, then introduce another paradigm for handling side-effect complexity: Redux Sagas. By the end of the chapter, you'll have at least one more tool in your state-management tool belt.

6.1 What are side-effects?

Redux handles data with as much purity as possible, but for most applications you can't avoid *side-effects*. Side-effects are required to interact with the world outside of your client application. Side-effects, in this context, are any interactions with the world beyond your Redux application. Most examples can be summarized as interactions with a server or local storage. For example, you may be storing an authorization token in a browser's sessionStorage, fetching data from a remote server, or recording an analytics event.

Where do we handle side-effects? We know that reducers must be pure functions to get the full benefits of Redux, and components ought to just dispatch action creators. That leaves action creators and middleware as options (figure 6.1).

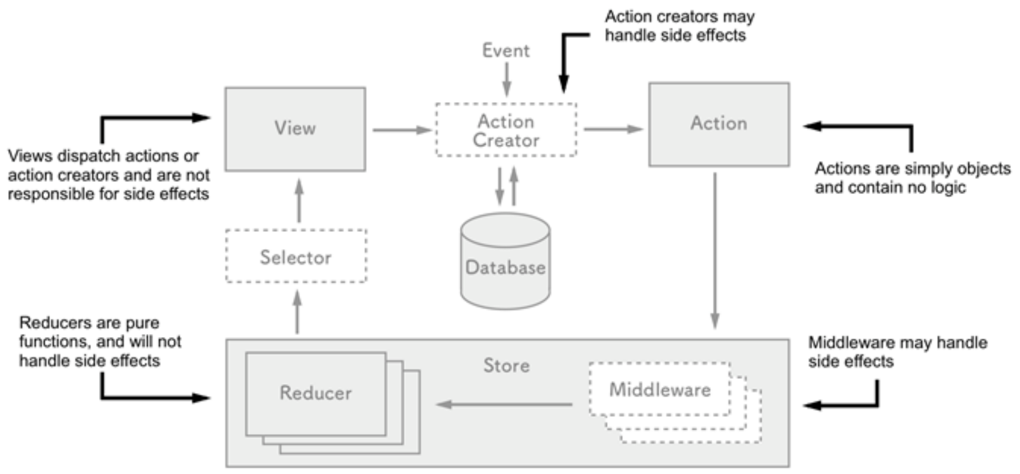

Action creators may handle side effects
Event
View
Action Creator
Action
Views dispatch actions or action creators and are not responsible for side effects
Actions are simply objects and contain no logic
Selector
Database
Reducers are pure functions, and will not handle side effects
Store
Reducer
Middleware
Middleware may handle side effects

Figure 6.1 Within the Redux architecture, only action creators and middleware should handle side-effects.

Conveniently, you already have experience using both action creators and middleware to handle side-effects. In the last chapter, you were exposed to an API middleware that handled AJAX requests. In chapter 4, we handled some simple side-effects within action creators by leveraging `redux-thunk`.

We should clarify: when asked where you want to handle side-effects, the answer is not restricted to either action creators or middleware. "Both" is an entirely reasonable answer! Remember, the `redux-thunk` package provides middleware to handle functions returned by action creators — an example of both action creators and middleware working together to manage side-effects.

Over the next couple of sections, we'll compare and contrast thunks with Redux Sagas, another pattern that leverages middleware. We'll start with thunks.

6.2 Revisiting thunks

You have some great exposure to thunks already, having used them to interact with a remote server to fetch a list of tasks or create a new one. To review, the Redux store and its reducers know what to do with actions. Actions just describe events — opening a modal, for example, can be the result of a clicked button dispatching an action of type `'OPEN_MODAL'`.

Things get more complicated when the contents of that modal need to be populated by an AJAX call. We could choose, for example, to dispatch an action only after the AJAX call returns, or dispatch several actions along the way to indicate loading progress. Again, the store deals in actions and is unequipped to handle functions or promises, so it's up to you to make sure that whatever reaches the store is an action.

When we need to perform asynchronous activities, `redux-thunk` makes it possible for an action creator to return a function in lieu of an action. Within the fetchTasks action creator, an anonymous function (thunk) is returned. The thunk middleware provides the `dispatch` and `getState` arguments, so the body of the function can view the contents of the current store and dispatch new actions to indicate loading, success or failure states. View listing 6.1 for a recap.

Listing 6.1 src/actions/index.js

```
export function fetchTasks() {
  return (dispatch, getState) => { // #A
    dispatch(fetchTasksRequest()); // #B
    ...
    dispatch(fetchTasksSuccess()); // #C
    ...
  }
}
```

#A The action creator returns a function, also known as a thunk
#B Within the thunk, more action creators may be dispatched
#C Based on the results of a side-effect, more dispatching may occur

6.2.1 Strengths

Thunks have a lot going for them. They're dead simple, ubiquitous in documentation, and powerful enough to be the only side-effect management tool you'll ever need.

SIMPLE

At the time of writing, the source code for `redux-thunk` is 14 lines — 11 if you don't count line breaks. Installing and using the library is intuitive and newcomer friendly. Additionally, you'll find excellent documentation in the GitHub repository and example usage in most Redux tutorials in the wild.

ROBUST

Although you'll learn about a few other side-effect management tools in this chapter, you can get the job done with thunks alone. They may be arbitrarily complex. Within a thunk, you're free to dispatch other action creators, make and respond to AJAX requests, interact with local storage, and so on.

GENTLE MIDDLEWARE INTRODUCTION

This is a tangential point, but worth mentioning. For most developers, `redux-thunk` is their first introduction to Redux middleware. Middleware tends to be one of the least accessible pieces of the Redux puzzle and implementing `redux-thunk` is about as gentle a middleware introduction as you can get. That education is a net gain for the community and for you the developer, because it helps demystify part of the architecture.

6.2.2 Weaknesses

There are tradeoffs with any tool, of course. The simplicity of thunks makes them something of a double-edged sword. Thunks are easy to write, but you're on your own to write advanced functionality.

VERBOSITY

Stuffing complex logic or multiple asynchronous events into a thunk can result in a function that's difficult to read or maintain. There's no magic behind the scenes or accompanying utility functions to aid with that, so it's up to you to manage.

TESTING

Testing is one of the clearest weaknesses of thunks. The pain generally comes from needing to import, create, and populate a mock store before you can make assertions about actions being dispatched. On top of that, you'll likely need to mock any HTTP requests.

6.3 Introducing sagas

The name is telling. *Sagas* are built to handle the hairy and winding story of your data. Using an ES6 feature, generators, the `redux-saga` package offers a powerful way to write and reason about complex asynchronous behavior. With a little new syntax, sagas can make asynchronous code as readable as synchronous code.

This chapter won't be an exhaustive look at `redux-saga` and all its use cases or features. The goal is to get you familiar enough with the basics to know whether your next feature could benefit from using a saga. The answer won't always be "yes."

A classic example of a good use case is a user login workflow. Logging in a user may require multiple calls to a remote server to validate credentials, issue or validate an authentication token, and return some user data. It's certainly possible to handle all this with thunks, but this realm is where sagas really shine.

6.3.1 Strengths

As we've alluded, sagas are not the answer to every problem. Let's explore what they are good for.

HANDLING COMPLEXITY AND LONG-RUNNING PROCESSES

Sagas helps you think about asynchronous code in a synchronous fashion. Instead of manually handling chains of promises and the spaghetti code that accompanies them, you're able to use an alternative control flow that results in cleaner code. Particularly challenging side-effects to handle are long-running processes. A simple example where you'll run into these types of problems is a stopwatch application. Its implementation with `redux-saga` is trivial.

TESTING

Sagas don't actually perform or resolve side-effects, they merely return descriptions of how to handle them. The execution is left to middleware, under the hood. Because of this, it's very straightforward to test a saga. Instead of requiring a mock store, you can simply test that a saga returns the correct side-effect description. We won't walk through a saga test in this chapter, but there are examples in the official documentation at https://redux-saga.js.org/docs/advanced/Testing.html.

6.3.2 Weaknesses

With great power comes great responsibility. Let's take a look at the tradeoffs involved with using `redux-saga`.

LEARNING CURVE

When bringing a newly-hired Redux developer onto your team, it's pretty safe to assume they're proficient with thunks. The same cannot be said for sagas, however. A common cost for using `redux-saga` is the time it takes to bring an unfamiliar developer or team of developers up to speed with using it. The use of generators and an unfamiliar paradigm can make for a steep learning curve.

HEAVY-HANDED

Put simply, `redux-saga` may just be overkill for some simple applications. As a rule of thumb, we prefer to introduce a saga only when enough pain points are experienced while using a thunk. Keep in mind that there are costs associated with including another package – developer onboarding and the additional file size, in particular.

Generators are well-supported by most modern browsers, but that hasn't always been the case. If your application needs to support older browser versions, you will want to consider the impact of including a generator polyfill, if required.

6.4 What are generators?

This book assumes you're pretty comfortable with the best-known ECMAScript 2015 syntax, but *generators* don't fall into that category. Generators enable powerful functionality, but the syntax is quite foreign and their use cases are still being discovered. For many React developers, `redux-saga` is their first introduction to generator functions.

Put simply, generators are functions that can be paused and resumed. The Mozilla Developer Network describes generators as "functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances." You may find it useful to think of them as background processes or subprograms.

6.4.1 Generator syntax

Generators look like any other function, except they are declared with an asterisk following the function keyword, as in the following example:

```
function* exampleGenerator() { … }
```

Note that when declaring a generator, the asterisk may come at any point between the `function` keyword and the function name. Each of the following are functionally the same:

```
function* exampleGenerator() { … }
function *exampleGenerator() { … }
function*exampleGenerator() { … }
function * exampleGenerator() { … }
```

This book standardizes on the first example, because it appears to be more popular in the wild and is the style preference chosen in the `redux-saga` documentation. Although we won't be writing any in this chapter, know that generators can also be anonymous:

```
function* () { … }
function *() { … }
function*() { … }
function * () { … }
```

Generators can *yield* results. The `yield` keyword can be used to return a value from the generator function. See listing 6.1 for an example.

Listing 6.1 Basic generator example

```
function* exampleGenerator() { // #A
   yield 42; // # B
   return 'fin';
}
```

#A The generator function is denoted with an asterisk
#B the `yield` keyword provides a return value from the generator function

What do you suppose might happen if you execute the function, `exampleGenerator()`? Go ahead and try it out in your terminal. Assuming you have Node.js installed, start the Node.js REPL by entering `node` into your terminal window, then write the function in listing 6.1 and execute it. Not quite what you expected, was it? The terminal output appears to be just an empty object.

6.4.2 Iterators

What the generator actually returns is a called an *iterator*. Iterators are objects, but they're not empty. They keep track of where they are in a sequence, and can return the next value in the sequence. Iterators have a `next` function that can be used to execute code within the generator up until the next `yield`, as shown in this example:

```
exampleGenerator();        //=> {}
exampleGenerator().next(); //=> { value: 42, done: false }
```

Notice the output of the next function. The result is an object with two keys, value and done. value contains the yielded content, 42. The done key has a value of false, indicating that the generator has more data to provide if called again. At this point, the generator function is effectively paused and waiting to be called on again to resume executing after the yield statement. Let's keep going:

```
exampleGenerator();        //=> {}
exampleGenerator().next(); //=> { value: 42, done: false }
exampleGenerator().next(); //=> { value: 42, done: false }
exampleGenerator().next(); //=> { value: 42, done: false }
```

Wait, what happened? Shouldn't value have been the string 'fin' and done returned true? Don't let this one bite you. Each time exampleGenerator is executed, a new iterator is returned. You'll need to store the iterator in a variable, then call next on the stored iterator. See listing 6.2 for an example.

Listing 6.2

```
const iterator = exampleGenerator(); // #A
iterator.next(); // { value: 42, done: false }
iterator.next(); // { value: 'fin', done: true } // #B
iterator.next(); // { value: undefined, done: true } // #C
```

#A Store the iterator created by the generator function
#B Having reached the return statement, done **flips to** true
#C Continuing to call next **returns an** undefined **value**

So far, you've seen yield and return statements used in a generator. We've learned that yield will return a value with done set to false. return does the same, with done set to true. A third option exists: throw. We won't use it in this chapter, but throw can be used to break out of a generator function in the case of an error. You can find more details about the throw method at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator/throw.

6.4.3 Looping with generators

We're willing to bet that the last infinite loop you wrote was an accident. It happens to the best of us. In the case of generators however, infinite loops are a viable usage pattern. The next listing is an example of an intentional infinite loop.

Listing 6.3 Infinite loop example

```
function* optimisticMagicEightBall() {
   while (true) { // #A
      yield 'Yup, definitely.';
   }
}
```

#A Create an infinite loop with a while (true) **block**

Now, we can answer an unlimited number of questions with our generator function, optimisticMagicEightBall. Each call of the next function on the iterator will return the affirmative answer, then pause, waiting for the loop to begin again.

It's also possible to compose generators, or in other words, use generators within generators. In this case (listing 6.4), looping through the count function will produce the numbers one through five in order as the count function becomes blocked until middleNumbers resolves. The yield* syntax is used to delegate out to another generator.

Listing 6.4 Composing generators example

```
function* count() {
   yield 1;
   yield 2;
   yield* middleNumbers(); // #A
   yield 5;
}

function* middleNumbers() {
   yield 3;
   yield 4;
}
```

#A The middleNumbers generator will complete before moving on to 5

Although these examples as contrived, be sure that they make sense before moving on. The redux-saga library makes heavy use of both.

6.4.4 Why generators?

Although capable, JavaScript has something of a reputation for its difficulty in managing series of asynchronous events. You may have heard the term "callback hell" used to describe a group of deeply nested, chained asynchronous functions regularly found in JavaScript codebases. Generators were created to offer an alternative control flow for asynchronous operations.

Generators are broadly applicable tools, and again, those applications are still being uncovered. Complex asynchronous operations and long-running processes are two of the most commonly cited opportunities to gain readability and maintainability by introducing a generator.

Generators are also a platform to build still more powerful or more developer-friendly tools. The popular ES7 feature, *async/await*, leverages generators to create another highly approachable way to handle complex asynchronous events. The doors are open for more intricate, domain specific libraries to be built, like redux-saga.

6.5 Implementing sagas

With an understanding of the fundamentals of generators, we're now ready to step into sagas. As an exercise, let's try refactoring one of Parsnip's thunks into a saga: the fetchTasks action creator. Visually, the final result will look identical to existing functionality, but under the hood, we'll be using an entirely new paradigm. To be clear, the net gain from this refactor won't be much: easier testing in exchange for code complexity. The value is in the academic exercise.

The first step is to install the package. Within your Parsnip app, add `redux-saga`.

```
npm install redux-saga
```

In the following sections, we'll walk through configuring an application to use the `redux-saga` middleware, then write our first saga. You'll discover that sagas and thunks are both a means to reach the same end, and for each feature you build, you may choose whichever tool fits the use case best.

6.5.1 Connecting saga middleware to the store

Sagas operate as middleware, and middleware gets registered in the store at the time of the store's creation. As a reminder, Redux's `createStore` function takes up to three arguments: reducers, initial state, and enhancers. In the last chapter, you learned that middleware gets applied in the last argument as a store enhancement. Here again, we'll register `redux-saga` as middleware.

In listing 6.5, we import and use the `createSagaMiddleware` factory function. Redux's `applyMiddleware` function takes a list of arguments, so we just can just add the saga middleware right alongside the thunk middleware. Remember, the order you list middleware determines the order an action will pass through them.

Listing 6.5 src/index.js

```
import createSagaMiddleware from 'redux-saga'; // #A
import rootSaga from './sagas'; // #B

const sagaMiddleware = createSagaMiddleware(); // #C

const store = createStore(
   reducer,
   composeWithDevTools(applyMiddleware(sagaMiddleware, thunk)) // #D
);

sagaMiddleware.run(rootSaga); // #E
```

#A `createSagaMiddleware` is the default export of `redux-saga`
#B `redux-saga` needs to know which saga (or sagas) to run
#C `createSagaMiddleware` is a factory function, used to create `sagaMiddleware`
#D Add the saga middleware to the list in `applyMiddleware`
#E Finally, initiate the saga with the `run` method on the `sagaMiddleware` instance

Again, there's no reason you can't use both thunks and sagas in the same application, and indeed, many applications do.

It may be helpful to think of sagas as subprograms, and the `run` function on the last line of the listing is required for the subprogram to begin watching for actions. Once the saga middleware is configured, we can `run` the top-level, or *root*, saga. In the next section, you'll get a chance to put a generator into production by writing a root saga.

6.5.2 Introducing the root saga

The code in the previous listing won't actually do anything for us until we write the root saga we've imported. Let's go just far enough to tell that we've configured `redux-saga` properly. Create a new file within the `src` directory and name it `sagas.js`. Within that file, write a `rootSaga` generator function and have it log a message to the console.

Listing 6.6 src/sagas.js

```
export function* rootSaga() { // #A
   console.log('rootSaga reporting for duty');
}
```

#A Denote generators with the asterisk following the `function` keyword

If everything went smoothly, you should see your message logged to the console after restarting the server. (If you have trouble with the json-server, revisit the setup instructions in appendix A.) So far so good! Let's pause here to explore what we've got and where we're headed.

After configuring the store to use saga middleware, we instructed the middleware to `run` the root saga. Because it's simpler to keep track of a single entry point, the root saga's role will be to coordinate all other sagas used in the application. When that root saga is implemented, we'll expect it to kick off sagas to run in the background, watching for and reacting to actions of specific types. As mentioned, we can and will use thunks at the same time, so each saga will listen for and respond to only specific actions. See figure 6.1 for a visual representation.

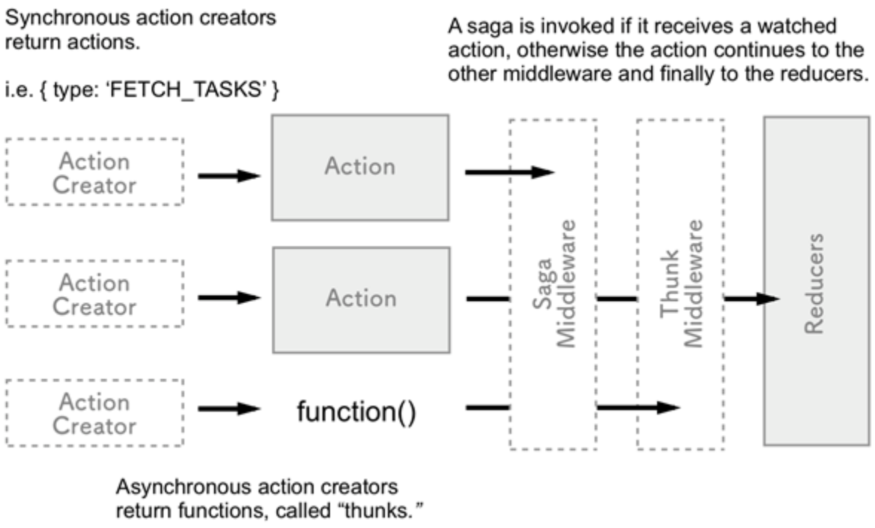

Synchronous action creators return actions.
i.e. { type: 'FETCH_TASKS' }
A saga is invoked if it receives a watched action, otherwise the action continues to the other middleware and finally to the reducers.
Action Creator
Action
Action Creator
Action
Action Creator
function()
Saga Middleware
Thunk Middleware
Reducers
Asynchronous action creators return functions, called "thunks."

Figure 6.1 Sagas will respond to actions of a specific type. If another action type or thunk is received, it will pass through the saga middleware unchanged.

We know that the order in which middleware is provided to the `applyMiddleware` function is the order that actions pass through them. Because we listed `sagaMiddleware` first, all dispatched values will pass through it before the thunk middleware. In general, a saga will react to an action, handling one or more side-effects, and eventually return another action to be processed by the reducers.

Ready for the big reveal? The first saga we'll write will be a replacement for the thunk that handles tasks fetching. The first thing we'll need to do is let the root saga know about this new saga.

> **POP QUIZ** **What are the three generator-specific methods and which is used to return a value from a generator without declaring it done? The three methods are `return`, `throw`, and `yield`. `yield` is the function that can return a value, but provide a `done` value of `false`.**

We're going to want to have our root saga yield each of the application's sagas. Could we use `return`? Sure, but only on the last line of the saga. The `redux-saga` documentation and examples choose to use only `yield`, so that's the pattern we'll stick with.

In the next listing, you'll see the root saga `yield` to another saga that will eventually watch for `'FETCH_TASKS'` actions. Sagas like `watchFetchTasks` are sometimes called *watchers*, because they infinitely wait and watch for particular actions. A common convention for spinning up multiple watchers is to *fork* them. We'll learn what that means in a minute. We only have one watcher to write so far, but for the sake of example, we'll add a second in listing 6.7 to demonstrate the conventions.

Listing 6.7 src/sagas.js

```
import { fork } from 'redux-saga/effects'; // #A

export function* rootSaga() {
   yield fork(watchFetchTasks); // #B
   yield fork(watchSomethingElse);
}

function* watchFetchTasks() { // #C
   console.log('watching!');
}

function* watchSomethingElse() {
   console.log('watching something else!');
}
```

#A Import `fork` from the `redux-saga/effects` package
#B Forking each watcher allows rootSaga to move on to the next one
#C Each watcher is also a generator

So what is `fork` doing here? When `rootSaga` executes, it's going to pause at every `yield` statement until the side-effect is completed. The `fork` method, however, allows `rootSaga` to move onto the next `yield` without a resolution. Each of these forks are said to be *non-blocking*. This implementation makes sense, because want to kick off all of the watchers at initialization, not just the first in the list.

6.5.3 Saga effects

In listing 6.7, you'll notice that we imported `fork` not from `redux-saga`, but from `redux-saga/effects`. `fork` is one of many methods made available to help you manage what are referred to as *effects*. One common misconception for newcomers is that the logic you write within a saga needs to do the processing of your side-effect, such as performing an AJAX request. That's not the case! Instead, the saga's role is to return a description of the logic needed in the form of an object. Figure 6.2 introduces the `call` method to illustrate this relationship.

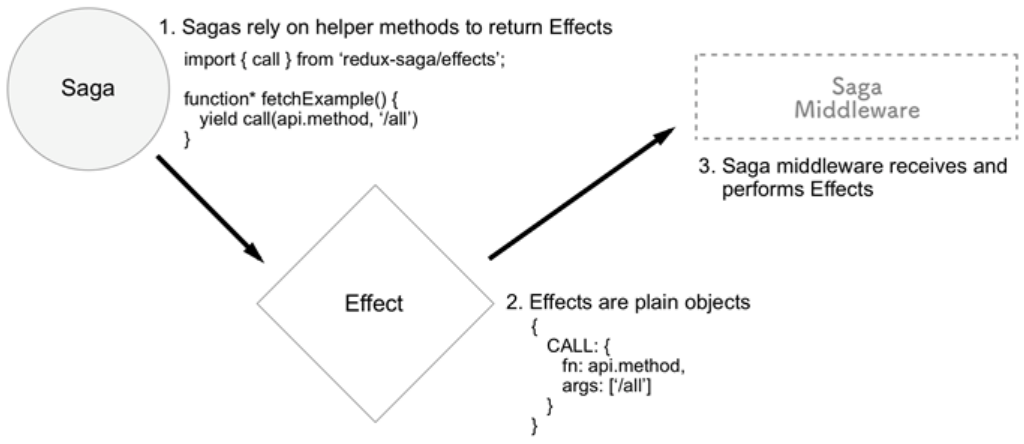

Saga
1. Sagas rely on helper methods to return Effects
import { call } from 'redux-saga/effects';
function* fetchExample() {
yield call(api.method, '/all')
}
Saga Middleware
3. Saga middleware receives and performs Effects
Effect
2. Effects are plain objects
{
CALL: {
fn: api.method,
args: ['/all']
}
}

Figure 6.2 Sagas return effects, which are instructions for the saga middleware to perform.

The `call` method used here is analogous to JavaScript's `call` function. We'll use it again shortly to specify the AJAX request we'll use to fetch the tasks. Once we use the `redux-saga` specific methods to generate an effect, the saga middleware will process and perform the required side-effects out of view. We'll learn more about these methods in the implementation of the task fetching next.

6.5.4 Responding to and dispatching actions

There's been a lot of talk about a watcher reacting only when the right action comes along. The method we're looking for is also among the group of helpers imported from `redux-saga/effects`: `take`. `take` is used to "wake up" and engage a saga when a particular action type arrives. Unlike `fork`, it *is* a blocking call, meaning that the infinite loop will halt while it waits for another action of type `'FETCH_TASKS_STARTED'` to come along. Listing 6.8 shows the basic usage.

Only after a `'FETCH_TASKS_STARTED'` action is dispatched and `take` is called will the `'started!'` log appear in your console. Notice that we've introduced an infinite loop into the saga to facilitate this feature. This technique shouldn't be alarming to you after reading the introduction to generators earlier in the chapter.

Listing 6.8 src/sagas.js

```
import { fork, take } from 'redux-saga/effects'; // #A
...
function* watchFetchTasks() {
   while (true) { // #B
      yield take('FETCH_TASKS_STARTED'); // #C
      console.log('started!');
   }
}
```

#A Import `take` from the effects package
#B Watchers use infinite loops to process actions as often as they're needed
#C `take` waits for a given action type before allowing the saga to proceed

If you're following along at home, you'll want to delete or comment out the related thunk feature. All you'll need to interact with this saga is to dispatch an action of type `'FETCH_TASKS_STARTED'`. You can accomplish this by exporting the `fetchTasksStarted` action and passing it to the dispatch in `componentDidMount` within the `App.js` component.

Now that we've got the basics of responding to actions covered, let's try out the other side of the coin, dispatching new ones. The method we're looking for this time is `put`. As an argument, `put` takes the action you'd like pass through to the remainder of the middleware and to the reducers. Let's bring the `call` method back into the picture and connect the remaining dots to complete the feature.

Listing 6.9 src/sagas.js

```
import { call, fork, put, take } from 'redux-saga/effects'; // #A
...
function* watchFetchTasks() {
   while (true) {
      yield take('FETCH_TASKS_STARTED');
      try {
         const { data } = yield call(api.fetchTasks); // #B
         yield put({ // #C
            type: 'FETCH_TASKS_SUCCEEDED',
            payload: { tasks: data }
         });
      } catch (e) {
         yield put({
            type: 'FETCH_TASKS_FAILED',
            payload: { error: e.message }
         });
      }
   }
}
```

#A Import each of the used helper methods
#B `call` is a blocking method used to specify the AJAX request
#C After a successful or unsuccessful request, `put` is used to dispatch an action

There's the whole fetch tasks feature replaced by a saga! The saga "wakes up" when `'FETCH_TASKS_STARTED'` is dispatched. It then waits for the middleware to perform the AJAX request, then dispatches a success or failure action with the results.

To verify it works on your machine, be sure to delete or comment out the related thunk, within the action creator `fetchTasks`, so you don't have two systems competing to process actions. All we'll need is a synchronous action creator to dispatch a `'FETCH_TASKS_STARTED'` action. See listing 6.10 for an example. Remember though, you already have the know-how for how to debug that situation yourself. You know that whichever order you provide middleware to `applyMiddleware` is the order that actions will move through them.

Listing 6.10 src/actions/index.js

```
...
function fetchTasks() {
   return { type: 'FETCH_TASKS_STARTED' }; // #A
}
...
```

#A The saga middleware will handle AJAX requests in response to this action type

If you compare this complete saga implementation of the fetch tasks feature with the thunk implementation, you'll notice they're not all that different. They're roughly the same amount of code and the logic looks quite similar. Introducing sagas undeniably added some complexity, though. Is it worth the learning curve?

We've mentioned, but not demonstrated, that sagas are easier to test than thunks. That's certainly worth something. It can also be really valuable to learn a new programming paradigm. At this point, however, if your answer is still "no," we wouldn't blame you. It's a hard sell to convince your development team to introduce a complex new tool without clearer value.

Fortunately, we've barely scratched the surface of what sagas may be useful for. As a quick example, let's say you wanted to cancel an unfinished, old request whenever a new one came in. In a thunk, this requires some extra labor, but `redux-saga/effects` provides a method, `takeLatest`, for this purpose. `takeLatest` replaces the use of `fork` in our root saga.

Listing 6.11 src/sagas.js

```
Import { call, put, takeLatest } from 'redux-saga/effects';
...
function* rootSaga() {
   yield takeLatest('FETCH_TASKS_STARTED', fetchTasks); // #A
}

function* fetchTasks() {
   try { // #B
      const { data } = yield call(api.fetchTasks);
      yield put({
         type: 'FETCH_TASKS_SUCCEEDED',
         payload: { tasks: data },
      });
   } catch (e) {
      yield put({
         type: 'FETCH_TASKS_FAILED',
         payload: { error: e.message },
      });
   }
}
...
```

#A `takeLatest` cancels old processes when a new one begins
#B No more infinite loop is required, as `takeLatest` continues to listen for the action type

Behind the scenes, `takeLatest` is creating a fork with some extra functionality. To provide its intended functionality, it will have to listen for every action of type `'FETCH_TASKS_STARTED'`. This keeps us from needing to do the same, so we can remove the infinite loop and `take` function from the `watchFetchTasks` saga. While we're at it, `watchFetchTasks` is no longer doing the watching, so we've tweaked the name to suggest that: `fetchTasks`.

That's the whole of it. Refresh your browser and you'll see identical results, but with more resource-conservative code underneath. To get that new feature, we deleted more code than we added. Sounds like a win-win to us.

6.6 Handling long-running processes

In the introduction of generators, we mentioned that handling long-running processes may be ideal use cases. Long-running processes may take many forms and one of the textbook educational examples is a timer or a stopwatch. We're going to run with this idea to add an additional feature to Parsnip, but with a little twist. In this section, we'll add a unique timer to each task, which begins when the task is moved to "In Progress." By the time we're done with the feature, it will look something like figure 6.3.

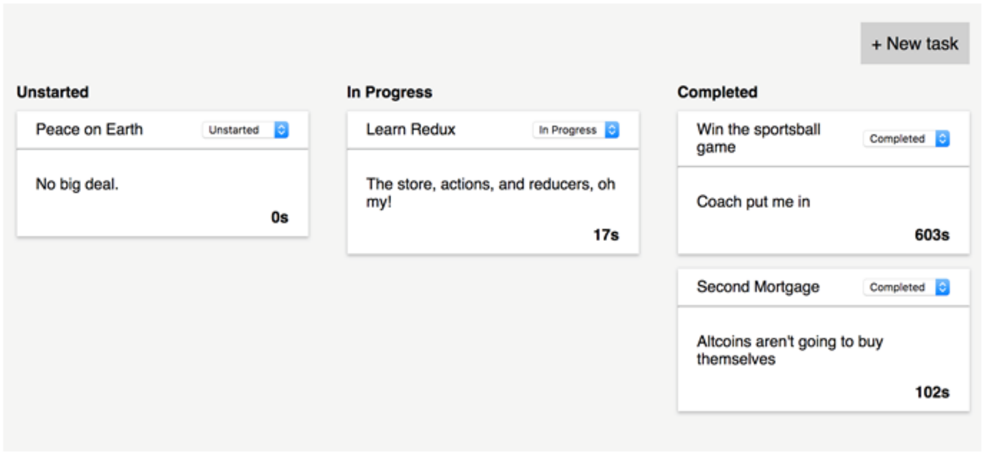


Figure 6.3 Tasks display a timer for how long they've been in progress.

6.6.1 Preparing data

We'll need the server to tell us at least what the timer starting value for each task is. To keep it simple, each task will have a timer key with an integer value, representing the number of seconds it has been in progress. See listing 6.12 for an abbreviated example. The specific numbers you choose are not important.

Listing 6.12 db.json

```
...
{
   “tasks”: [
      ...
      {
         “id”: 2,
         “title”: “Peace on Earth”,
         “description”: “No big deal.”,
         “status”: “Unstarted”,
         “timer”: 0 // #A
      }
      ...
   ]
}
...
```

#A Give each task a `timer` key and number value

Be sure to add the `timer` key to every task record. Remember, this is JSON formatting, so don’t forget your quotes around `timer`. Number values do not require them.

6.6.2 Updating the user interface

Next up is to display the new timer data in the tasks. This calls for a small addition to some already straightforward React code, so we aren’t going to publish the entire component here. Within the `Task` component, render the timer property somewhere within the body of the task. Use the following code as an example and style it as you please. The “s” is just an abbreviation for seconds.

```
<div className="task-timer">{props.task.timer}s</div>
```

At this point, the timer values you entered in the `db.json` file should be visible in the UI. The parent components of the `Task` component already make the value available, so no additional configuration is required.

6.6.3 Dispatching an action

Before we get to the saga, we’ll need to dispatch an action to get its attention with. The goal is to turn the timer on whenever a task is moved into “In Progress.” Mull it over: where would you want to dispatch a `‘TIMER_STARTED’` action?

We can piggyback onto existing logic to facilitate this. The action creator `editTask` already handles moving an action into each status column, so we can just dispatch an additional action whenever the destination is the “In Progress” column. See listing 6.13 for an example implementation.

Listing 6.13 src/actions/index.js

```
...
function progressTimerStart(taskId) {
   return { type: 'TIMER_STARTED', payload: { taskId } }; #A
}

export function editTask(id, params = {}) {
   return (dispatch, getState) => {
      const task = getTaskById(getState().tasks.tasks, id);
      const updatedTask = {
         ...task,
         ...params,
      };
      api.editTask(id, updatedTask).then(resp => {
         dispatch(editTaskSucceeded(resp.data));
         if (resp.data.status === 'In Progress') { // #B
            dispatch(progressTimerStart(resp.data.id));
         }
      });
   };
}
```

#A Dispatch the action that the saga will be listening for
#B Add an additional dispatch if the task moved into 'In Progress'

We need to pass the ID of the task to the action. When we have multiple timers incrementing at once, we'll need to know exactly which task to increment or pause.

6.6.4 Writing a long-running saga

If everything in this chapter is starting to click for you, you may already have an idea of where you'd like to go with this saga. One strategy is to listen for actions of type `'TIMER_STARTED'`, then increment the timer value once per second within an infinite loop. It's a good place to start!

We can begin by registering a `handleProgressTimer` saga with the root saga. This is an opportunity to introduce one more alternative to `fork`. `takeLatest` makes a lot of sense when you want to throttle API requests to a remote server. Sometimes you want to let everything through, though. The method we're looking for is `takeEvery`.

The implementation of `handleProgressTimer` will introduce one new method, but it's very self-explanatory: `delay`. `delay` is a blocking method, meaning that the saga will pause at its location until the blocking method resolves. You can see a complete implementation in listing 6.14.

You'll notice that we're importing `delay` not from `redux-saga/effects`, but instead from `redux-saga`. `delay` does not help produce an Effect object, so it doesn't reside alongside the other Effect helpers. We'll lean on the `call` method to produce the Effect and pass in `delay` as an argument, just like we did when making an API request.

Listing 6.14 src/sagas.js

```
import { delay } from 'redux-saga'; // #A
import { call, put, takeEvery, takeLatest } from 'redux-saga/effects'; //#B

export default function* rootSaga() {
   yield takeLatest('FETCH_TASKS_STARTED', watchFetchTasks);
   yield takeEvery('TIMER_STARTED', handleProgressTimer); // #C
}

function* handleProgressTimer({ payload }) { // #D
   while (true) { // #E
      yield call(delay, 1000); // #F
      yield put({
         type: 'TIMER_INCREMENT',
         payload: { taskId: payload.taskId }, // #G
      });
   }
}
...
```

#A Add the `takeEvery` method to the list of imports
#B Every time 'TIMER_STARTED' is dispatched, invoke the `handleProgressTimer` function
#C Add the `delay` method from `redux-saga`
#D Action properties are available as arguments
#E The timer runs infinitely while in progress
#F `delay` is used to wait one second (1000ms) between increments
#G The task ID is passed to the reducer to find the task to increment

Without extra configuration, the `take`, `takeEvery`, and `takeLatest` methods will pass the action through to the function or saga you provide. `handleProgressTimer` can access the `taskId` from the action payload to eventually specify which task to update.

6.6.5 Handling the action in the reducer

Every second, the saga will dispatch a 'TIMER_INCREMENT' action. By now, we know that it is the reducer's job to define how the store will update in response to that action. We'll want to create a case statement for handling the action within the tasks reducer. Much of this code is identical to the code for handling the 'EDIT_TASK_SUCCEEDED' action in the same reducer. The goal is simply to find the desired task, update its timer value, then return the list of tasks.

Listing 6.15 src/reducers/index.js

```
...
case 'TIMER_INCREMENT': { // #A
   const nextTasks = state.tasks.map(task => { // #B
      if (task.id === action.payload.taskId) {
         return { ...task, timer: task.timer + 1 }; // #C
      }
      return task;
   }

   return { ...state, tasks: nextTasks }; // #D
}
...
```

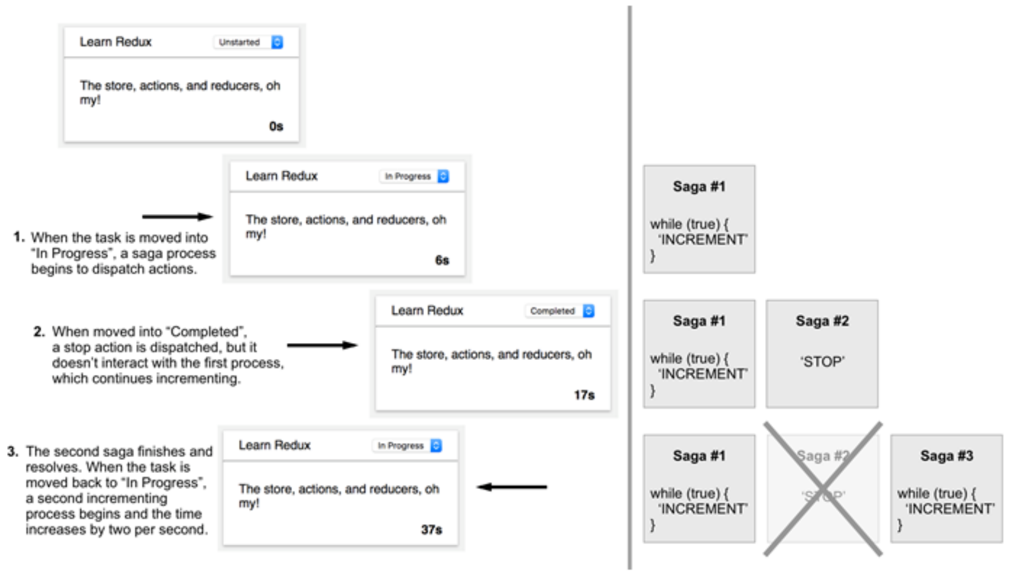
#A Add the new action type to the tasks reducer
#B Map over the existing tasks to create the updated versions
#C If the task ID matches, increment the task's timer
#D Return the updated tasks

That wraps up the implementation! If you try it out on your local machine, you should see the timer begin to tick up on any task that gets moved into the "In Progress" column. Multiple tasks increment uniquely, as you would expect. Not bad, right?

Don't celebrate quite so quickly though; our solution is a short-sighted one. In the exercise for this chapter, you'll be asked to implement the ability to stop the timer when a task in progress is moved to "Completed" or back to "Unstarted." If you attempt to add this functionality to our existing saga, you'll eventually be left scratching your head as to why the tasks aren't responding to `'TIMER_STOPPED'` actions or breaking out of the increment loop.

6.6.6 Using channels

We'll save you the head-scratching trouble and reveal what's actually going on here. `takeEvery` is starting a new process for each `'TIMER_STARTED'` action it receives. Each time a task is moved into "In Progress," a separate process begins dispatching `'TIMER_INCREMENT'` actions. That's a good thing. We want each task to increment individually. However, if a `'TIMER_STOPPED'` action were to come through the `handleProgressTimer` saga, it too would start a new process, separate from the one busy incrementing the timer. That is to say, there's no way to stop the incrementing process if we can't target it specifically. See figure 6.4 for an illustration.



Learn Redux
Unstarted
The store, actions, and reducers, oh my!
0s
1. When the task is moved into "In Progress", a saga process begins to dispatch actions.
Learn Redux
In Progress
The store, actions, and reducers, oh my!
6s
Saga #1
while (true) { 'INCREMENT' }
2. When moved into "Completed", a stop action is dispatched, but it doesn't interact with the first process, which continues incrementing.
Learn Redux
Completed
The store, actions, and reducers, oh my!
17s
Saga #1
while (true) { 'INCREMENT' }
Saga #2
'STOP'
3. The second saga finishes and resolves. When the task is moved back to "In Progress", a second incrementing process begins and the time increases by two per second.
Learn Redux
In Progress
The store, actions, and reducers, oh my!
37s
Saga #1
while (true) { 'INCREMENT' }
Saga #2
'STOP'
Saga #3
while (true) { 'INCREMENT' }

Figure 6.4 The takeEvery method spawns a new saga process for each action.

In the figure, we can see that by the time we've moved the task into "In Progress" twice, we would have two separate loops incrementing the timer every second. That's clearly not what we had in mind. If we were to use `takeLatest` instead of `takeEvery`, only one task could increment its timer at a time. Ultimately, what we want to achieve is the functionality of `takeLatest`, with a separate process per task. Simply stated, if we start a process, we need to be able to stop the same process. So, that's exactly the functionality we'll build.

To build this feature, we're going to leverage another `redux-saga` utility called a *channel*. From the official documentation, channels are just "objects used to send and receive messages between tasks." The *tasks* they reference are what we've been referring to as *processes*, to avoid naming confusion with our Parsnip tasks. Essentially, we'll be using channels as a way to give a name to a saga process, so that we can revisit the same channel again. If the language is confusing, the code should help to clarify it.

What we'll want to accomplish is to create a unique channel for each Parsnip task that starts a timer. If we keep a list of channels, we can then send a `'TIMER_STOPPED'` action to the correct channel when the task is moved to "Completed." We're going to go off the beaten path a little by creating a helper function to manage these channels.

We'll name the function `takeLatestById`, and what it'll do is send each action to the correct process. If a process doesn't already exist, a new one will be created. Listing 6.16 offers one implementation.

There's a lot of new code here, but there shouldn't be anything wildly surprising. `takeLatestById` is a generic helper function that is used to create rediscoverable processes. The function checks to see if a channel exists for a task, and if not, creates one and adds it to the mapping. After adding to the mapping, the new channel is immediately instantiated and the final line in the listing dispatches the action to the new channel. Conveniently, nothing about the `handleProgressTimer` function needs to change.

Listing 6.16 src/sagas.js

```
import { channel, delay } from 'redux-saga'; // #A
import { call, put, take, takeLatest } from 'redux-saga/effects'; // #B

export default function* rootSaga() {
   yield takeLatest('FETCH_TASKS_STARTED', watchFetchTasks);
   yield takeLatestById('TIMER_STARTED', handleProgressTimer); // #C
}

function* takeLatestById(actionType, saga) {
   const channelsMap = {}; // #D

   while (true) {
      const action = yield take(actionType);
      const { taskId } = action;

      if (!channelsMap[taskId]) {
         channelsMap[taskId] = channel(); // #E
         yield takeLatest(channelsMap[taskId], saga); // #F
      }

      yield put(channelsMap[taskId], action); // #G
   }
}
```

```
...
```

#A Add channel to the list of imports
#B Add take back to the list of effect helper imports
#C Have the root saga initiate the helper function
#D Store a mapping of created channels
#E If a task doesn't have a channel, create one
#F Create a new process for that task
#G Dispatch an action to the specific process

If you start the application on your machine, you should experience the same functionality with one important difference. Moving a task into "In Progress" twice should produce only one saga process, instead of two. The helper function has set up a `takeLatest` watcher for saga being manipulated. The second time the task is moved to "In Progress," the `takeLatest` function cancels the first process and starts a new one, never producing more than one increment loop per task. That's more like it!

6.7 Exercise

Your turn! All the scaffolding is set up for you to write a few lines of code to add the stop timer functionality to Parsnip. Specifically, what we want is for a task's timer to stop anytime it moves from "In Progress" to one of the other columns: "Unstarted" or "Completed."

Just because there's only a few lines to write, doesn't mean it's easy though. Completing this exercise will be a good way to prove to yourself that everything we've done so far is beginning to make sense. We'll give you one hint to get started: functions like `take` can be configured to accept and respond to more than one action type at a time. To do so, you can pass it an array of action type strings as the first argument.

6.8 Solution

Were you able to figure it out? The first step is to figure out when to dispatch the '`TIMER_STOPPED`' action. Right after the logic to determine whether to dispatch the 'TIMER_STARTED' action is a reasonable opportunity. See listing 6.17 for an example.

Listing 6.17 src/actions/index.js

```
...
export function editTask(id, params = {}) {
   ...
   api.editTask(id, updatedTask).then(resp => {
      dispatch(editTaskSucceeded(resp.data));

      if (resp.data.status === 'In Progress') {
         return dispatch(progressTimerStart(resp.data.id)); // #A
      }

      if (task.status === 'In Progress') { // #B
         return dispatch(progressTimerStop(resp.data.id));
      }
   }
}

function progressTimerStop(taskId) {
   return { type: 'TIMER_STOPPED', payload: { taskId } }; // #C
}
```

#A Don't forget the `return` keyword on the start dispatch
#B If the task, prior to updating, was "In Progress", stop the timer
#C Return the new `'TIMER_STOPPED'` action with the task ID

The part that may be confusing here is that within the `editTask` function, `task` refers to the task before it was updated, and `resp.data` refers to the updated task returned from the AJAX request. So, to start the timer, we check if the new task is in progress. If so, `editTask` dispatches another action and finishes there. We never move on to the check if the timer needs to be stopped. If the updated task is not in progress, but the original task was, then stop the timer.

Next, we need to handle the new `'TIMER_STOPPED'` action in the saga. This is going to be even less code than you expected.

Listing 6.18 src/sagas.js

```
...
export default function* rootSaga() {
   yield takeLatest('FETCH_TASKS_STARTED', watchFetchTasks);
   yield takeLatestById(['TIMER_STARTED', 'TIMER_STOPPED'], handleProgressTimer); // #A
}

handleProgressTimer({ payload, type }) { // #B
   if (type === 'TIMER_STARTED') { // #C
      while (true) {
         ...
      }
   }
}
...
```

#A Pass both action types in an array to the helper function
#B Add type to the list of destructured arguments
#C Wrap all the function's logic in a conditional statement that executes if type is 'TIMER_STARTED'

Seriously, that's it. Once we begin watching for both actions, the infrastructure we've already written will handle the rest. The helper function accepts the array of action types and passes the pattern into its `take` function. From that point on, start and stop actions will find themselves in the correct process, executing the code within the `handleProgressTimer` function.

Implementing the stop function doesn't require any other modifications to the function, because all we need it to do is not execute the increment logic. The 'TIMER_STOPPED' function bypasses the infinite loop and moves on to the reducers, eventually becoming visible in the Redux DevTools.

6.9 Additional side-effect management strategies

Thunks and sagas are the most popular side-effect management tools around, but in the open-source world, there's something for everyone. We'll discuss a few additional tools, but know that there are more options waiting for you to discover them.

6.9.1 Asynchronous functions with async/await

A feature introduced in ES7, async/await, has quickly found itself in many Redux codebases, often working in tandem with thunks. Using the feature is a very natural step to take for those already comfortable with thunks. You'll notice the control flow feels similar to sagas, and that can be explained by the fact that async/await uses generators under the hood. See listing 6.19 for an example within a thunk.

Listing 6.19 An async/await example

```
export function fetchTasks() {
   return async dispatch => { // #A
      try { // #B
         const { data } = await api.fetchTasks(); // #C
         dispatch(fetchTasksSucceeded(data));
      } catch (e) {
         dispatch(fetchTasksFailed(e));
      }
   }
}
```

#A Add the async keyword to the anonymous function
#B One error handling strategy is to use a try/catch block
#C Use the await keyword to block the function until the value returns

Why would you choose async/await? It's simple, powerful, and easy to learn. Why would you choose sagas? You need the functionality provided by the advanced features, the ease of testing is important to you, or maybe you just prefer the paradigm.

6.9.2 Handling promises with redux-promise

The `redux-promise` library is another tool maintained by Redux co-creator, Andrew Clark. The premise is simple: whereas `redux-thunk` allows action creators to return functions, `redux-promise` allows action creators to return promises.

Also like `redux-thunk`, `redux-promise` provides middleware that can be applied during the creation of the store. There are some functional nuances to using promises over thunks, or both in tandem, but choosing is largely a matter of style preference. The package is available on GitHub here: https://github.com/acdlite/redux-promise.

6.9.3 Redux-loop

This library diverges from what you've learned so far about the Redux architecture. Ready for some rule-breaking? `redux-loop` permits the use of side-effects within reducers. Stressful, we know.

As you may recall, Redux draws inspiration from several sources, Elm among the most influential. Within the Elm architecture, reducers are powerful enough to handle synchronous and asynchronous state transitions. This is achieved by having the reducers describe not just the state that should update, but also the *effects* that cause it.

Redux, of course, did not inherit this pattern from Elm. Redux can handle only synchronous transitions; all side-effects must be resolved (by hand or by middleware) before they reach the reducers. However, the Elm effects pattern is available for use in Redux using `redux-loop`, a minimal port of the functionality. You can find the package on GitHub here: https://github.com/redux-loop/redux-loop.

6.10 Summary

There are a number of options for handling complex side-effects. The severity of the complexity will help you determine which tool is the right one for the job, but you'll probably develop some favoritism as you find strategies that make the most sense to you and your programming style.

`redux-saga` is just another tool for your tool belt. Some folks will choose to replace all their thunks with sagas, while others will decide there's just too much magic going on there. Again, you can still accomplish it all just by using thunks. In practice, many applications succeed by finding a happy medium.

In this chapter, you learned:

- Thunks are sufficient to manage side-effects of any size
- Introducing sagas may help to tame especially complex side-effects
- Sagas are built using functions that can be paused and resumed, called generators
- Sagas produce effects, descriptions of how to handle side-effects

With `redux-saga` under our belt, we can confidently say we've got a handle on some of the messiest corners of Redux applications. We can move on to further optimizations elsewhere in the codebase by learning how to use selectors, like `reselect`, in the next chapter.

A Installation

This appendix covers

- **Setting up a server**
- **Installing axios**
- **Configuring redux-thunk**
- **Updating the Redux DevTools**

A.1 Setting up a server

Before we can start dispatching async actions, we need do a few things:

- Install and configure `json-server`, a popular tool for generating REST APIs quickly. This is a Redux book after all, so we don't want to expend too much energy writing a fully featured backend. `json-server` allows us to specify a set of resources (like tasks), and will create a standard set of endpoints for us to use.
- Install `axios`, a popular AJAX library. There's no shortage of popular AJAX libraries, including the new native browser API `window.fetch`, but `axios` is our choice due a solid feature set and a straightforward API.
- Install and configure `redux-thunk`. We'll have to add some middleware to the Redux store, which is simpler than it might sound.

A.1.1 Installing and configuring json-server

First things first, install `json-server` globally by running the following command in a terminal window:

```
npm install --global json-server
```

Next, create a `db.json` file and add the following contents.

Listing 1.1 db.json

```
{
  "tasks": [
    {
      "id": 1,
      "title": "Learn Redux",
      "description": "The store, actions, and reducers, oh my!",
      "status": "Unstarted"
    },
    {
      "id": 2,
      "title": "Peace on Earth",
      "description": "No big deal.",
         "status": "Unstarted"
       },
    {
      "id": 3,
      "title": "Create Facebook for dogs",
      "description": "The hottest new social network",
      "status": "Completed"
    }
  ]
}
```

Finally, start the server with the following command. Note that we're specifying the server to run on port 3001, because the Create React App development server is already running on port 3000.

```
json-server --watch db.json --port 3001
```

That's it! We'll start to use each of the newly available endpoints as we add more functionality throughout the chapter.

A.2 Installing axios

With a server configured, we need to install one more package, `axios`. `axios` is one of the many options for making AJAX work a bit more pleasant, and it will be our library of choice for making HTTP requests from the browser throughout the rest of the book. Make sure you're in the parsnip directory and install axios via npm:

```
npm install -P axios
```

If you see the standard npm success output, you're all set!

A.3 redux-thunk

Install `redux-thunk` via npm from the terminal by running the following command:

```
npm install -P redux-thunk
```

Add the middleware via the `applyMiddleware` function exported by Redux.

Listing 1.2 src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore, applyMiddleware } from 'redux';  //#A
import { Provider } from 'react-redux';
import thunk from 'redux-thunk';  //# B
import { tasks } from './reducers';
import App from './App';
import './index.css';

const store = createStore(
  tasks,
  applyMiddleware(thunk)  //#C
);

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

#A Import `applyMiddleware` from `redux`
#B Import the middleware from `redux-thunk`
#C Apply the middleware during store creation

Before we get off to the races, there's one more important update to make regarding the Redux DevTools.

A.4 Configuring the Redux DevTools

Now that we've introduced some middleware, the old `devToolsEnhancer` function is no longer going to do the trick. We'll import another method from the same DevTools library that can accommodate middleware, called `composeWithDevTools`.

Listing 1.3 src/index.js

```
import { composeWithDevTools } from 'redux-devtools-extension'; // #A
...
const store = createStore(
  tasks,
  composeWithDevTools(applyMiddleware(thunk)) // #B
);
```

#A import composeWithDevTools
#B wrap the applyMiddleware function

Now we're ready to return to the action! Don't forget to use the DevTools Chrome extension to see each action come through the system. The API story resumes in chapter 4.