SAMPLE CHAPTER

Redux
IN ACTION

Marc Garreau
Will Faurot
Foreword by Mark Erikson

MANNING

Redux in Action

by Marc Garreau
Will Faurot
Foreword by Mark Erikson

Chapter 1

brief contents

1 Introducing Redux

This chapter covers

- Defining Redux
- Understanding the differences between Flux and Redux
- Using Redux with React
- Introducing actions, reducers, and the store
- Learning when to use Redux

If you hop into any React web application in 2018, 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. A few years ago, Redux had yet to be created and React enjoyed an excited and blossoming user base. Early adopters of React believed that they'd found the best solution yet to the view layer—the "V" of the MVC (Model-View-Controller) 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. Eventually, Redux settled the debate.

Throughout the course of this book, we'll explore Redux and its ecosystem through the lens of a React application. As you'll learn, Redux can be plugged into JavaScript applications of all flavors, but React is an ideal playground for a few rea-

sons. Chief among those reasons: Redux was created in the context of React. You're most likely to encounter Redux within a React application, and React is agnostic about how you manage the data layer of your application. Without further ado, let's jump in.

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 using 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 wasn't 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 a 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 makes live application updates while maintaining existing state, and his nascent Redux developer tools enable 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 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 views, 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 scrapped the two-way data-binding model in favor of a unidirectional data flow. Instead of permitting each view to interact with its 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.

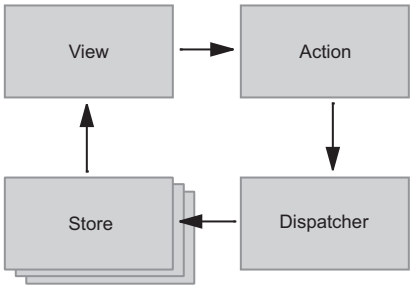

Figure 1.1 Flux specifies that data must flow in a single direction.

1.2.1 Actions

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

1.2.2 Dispatcher

All data flow in a Flux application is funneled through a single dispatcher. The dispatcher itself has 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 begins to receive actions. When it receives an action type that it cares about, the store updates 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 from which they want to display data. 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 these alternative implementations can be found in section 1.6.

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" (https://redux.js.org/). It's a standalone library, but it's 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 applications. Redux divides the responsibilities of state management into a few separate units:

- The store holds all your application state in a single object. (We'll commonly refer to this object as the state tree.)
- The store can be updated only with 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, then return the next state after applying any updates.

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

Redux enjoys the benefits of a predictable data flow from the Flux architecture, but it has 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. You'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 using bindings, as shown in figure 1.2.

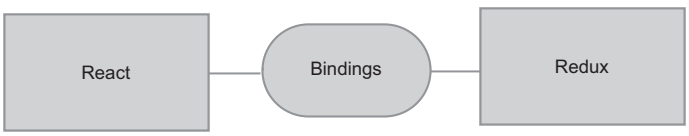

Figure 1.2 Redux isn't part of any existing framework or library, but additional tools called bindings connect Redux with React. Over the course of the book you'll use the react-redux package for this.

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

Although this book is fundamentally about Redux, our treatment of it is closely tied to React. Redux is a 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 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 several other JavaScript frameworks.

1.3.2 The three principles

You 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 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's 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 to flip back and forth through snapshots of application state, is what inspired the creation of Redux in the first place.

STATE IS READ-ONLY

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 don't 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 section.

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 mutates the existing state while producing the new one, you may end up with an erroneous new state, but you also lose the predictable transaction log that each new action should 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

We've touched briefly upon topics such as actions, reducers, and the store, but in this section, we 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 you'll have plenty of time in later chapters to apply the concepts you're about to explore.

Modern web applications are ultimately about handling events. They could be initiated by a user, such as navigating to a new page or submitting a form. Or they could be initiated by another external source, such as a server response. Responding to events usually involves updating state and re-rendering with that updated state. The more 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 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 to 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. Figure 1.3 shows a quick mockup of a user profile page, which may or may not take its inspiration from Twitter.

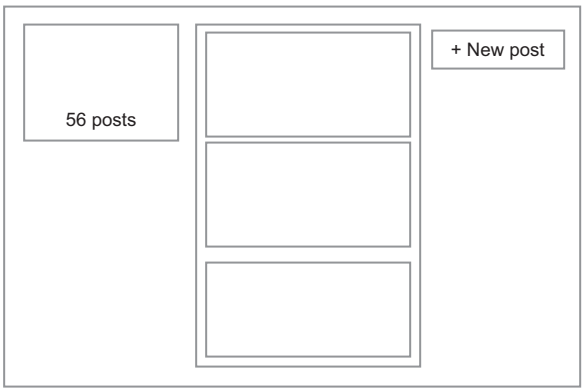

Figure 1.3 A simple mockup of a profile page. This page is backed by two main pieces of data: the total post count and the list of post objects in the user's activity feed.

The following distinct steps are involved in handling an event such as 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 similar to 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. You'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 such as middleware and selectors, which we'll cover in depth in later chapters.

ACTIONS

You want to do two things in response to a user submitting a new post: add the post to the user's activity feed and increment their total post count. After the user submits, you'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. You 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 your case, you only need one field: the contents of the message we want to post. The name "payload" is only a popular convention.

Actions have the advantage of serving as audits, which keep 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 describe an event; they don't know and don't care what happens downstream. Somewhere down the road you'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're simple functions that take your current state and an action as arguments, and return the next state. See figure 1.5.

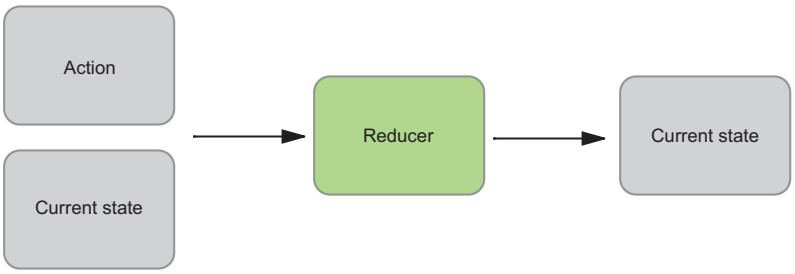

Figure 1.5 An abstract representation of a reducer's function signature. If this diagram looks simple, that's because it is! Reducers are meant to be simple functions that compute a result, making them easy to work with and test.

Reducers are typically easy to work with. Similar to 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. Figure 1.6 shows how our reducer might update the list of posts and the total post count.

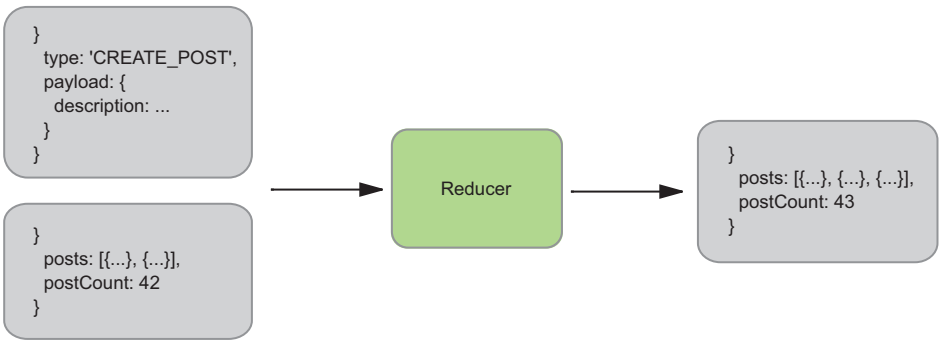

Figure 1.6 Visualizing a reducer hard at work. It accepts as input an action and the current state. The reducer's only responsibility is to calculate the next state based on these arguments. No mutations, no side-effects, no funny business. Data in, data out.

You're focusing on a single event in this example, which means you 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 they 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 this case). View bindings provided by react-redux will allow you to receive updates from the store and respond to them in your 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 (you care specifically about the components that make up your profile page). See figure 1.7.

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 your user adds another post, you'd follow the same exact flow. The view dispatches 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 several of the most important building blocks, let's look at a more comprehensive diagram of the Redux architecture. Several pieces will be unfamiliar now, but we'll revisit this diagram (figure 1.8) repeatedly throughout this book, and over time, we'll fill in each of those gaps.

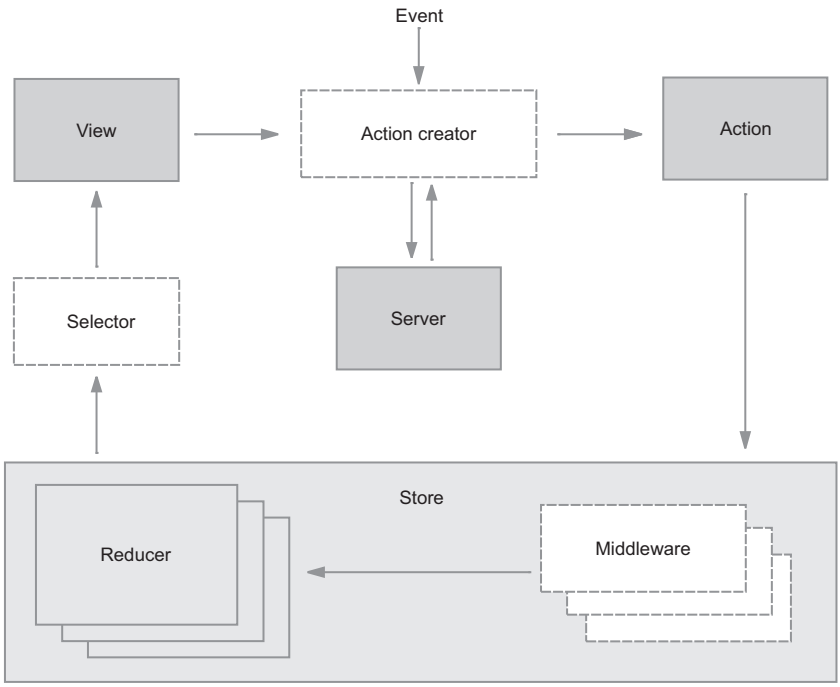

Figure 1.8 This diagram will anchor your understanding of the elements of Redux as you move forward. At this point, we've talked about actions, reducers, the store, and views.

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 updates, the views will be made aware that there's new data to render. That's the whole cycle! Items in figure 1.8 with a dashed border (action creators, middleware, and selectors) are optional, but powerful, tools in a Redux architecture. We 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 to 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. If 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 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 uses Jest and Enzyme, but whichever JavaScript testing libraries your organization prefers will do the trick.

1.4.4 Learning curve

Redux is a 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 several functional programming patterns. This will be new territory for certain 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—under 7KB when minified. Checkmate.

1.5 When should I use Redux?

We've hit 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? Teams use a few common scenarios 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 performing these tasks in React, but eventually you have 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 upfront.

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 8 of them have received more than 1,000 stars on GitHub. This highlights an important era in React's history—the Flux architecture was a groundbreaking idea that spurred 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." You'll find great documentation about the Flux architecture pattern 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 functional reactive programming ideas to the Flux architecture by ripping out the single Dispatcher in favor of giving each action the ability to dispatch itself. 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 uses 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, you 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 part of the terminology in functional reactive programming can be intimidating, the features are 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's 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 was originally implemented with Redux under the hood, but now offers additional configurability.

While it's possible to bring in both Redux and a GraphQL client to manage the same application's state, you may find the combination to be overly complex and unnecessary. Although GraphQL clients handle data fetching from a server and Redux is more general-purpose, there's overlap in usage between the packages.

Summary

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

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

Key points you've learned

- Redux state is stored in a single object and is the product of pure functions.
- For the price of 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.

Redux IN ACTION

Garreau • Faurot

With Redux, you manage the state of a web application in a single, simple object, practically eliminating most state-related bugs. Centralizing state with Redux makes it possible to quickly start saved user sessions, maintain a reliable state history, and smoothly transfer state between UIs. Plus, the Redux state container is fully programmable and integrates cleanly with React and other popular frameworks.

Redux in Action is an accessible guide to effectively managing state in web applications. Built around common use cases, this practical book starts with a simple task-management application built in React. You'll use the app to learn the Redux workflow, handle asynchronous actions, and get your hands on the Redux developer tools. With each step, you'll discover more about Redux and the benefits of centralized state management. The book progresses to more-complex examples, including writing middleware for analytics, time travel debugging, and an overview of how Redux works with other frameworks such as Angular and Electron.

What's Inside

- Using Redux in an existing React application
- Handling side effects with the redux-saga library
- Consuming APIs with asynchronous actions
- Unit testing a React and Redux application

For web developers comfortable with JavaScript and React.

Marc Garreau has architected and executed half a dozen unique client-side applications using Redux. **Will Faurot** is a mentor for Redux developers of all skill levels.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/redux-in-action

"Comprehensive, practical, does a great job of teaching many key topics for real-world Redux apps."
—From the Foreword by Mark Erikson
Redux co-maintainer

"The authors do a wonderful job of making the material compelling."
—Jeremy Lange, Sertifi

"Take control of your application state with expert Redux advice."
—Ian Lovell
Parmenion Capital Partners

"A perfect example of a book beating online resources."
—Jose San Leandro, OSOCO

ISBN-13: 978-1-61729-497-6
ISBN-10: 1-61729-497-7
54499
9 781617 294976

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]