Intro to React & Redux

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.

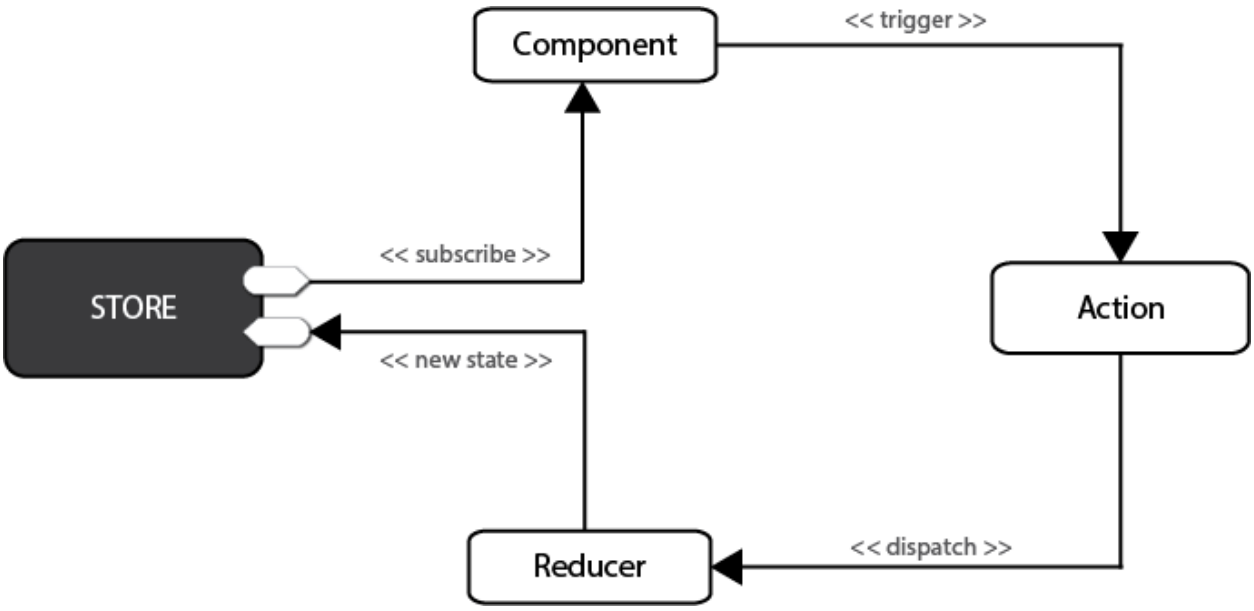
Let's consider a use case involving the authentication status of a user. Your product manager tells you that when a user is logged into an ecommerce store, the navigation bar should display the user's avatar image, the store should display items nearest to the user's zip code first, and the newsletter 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.

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?

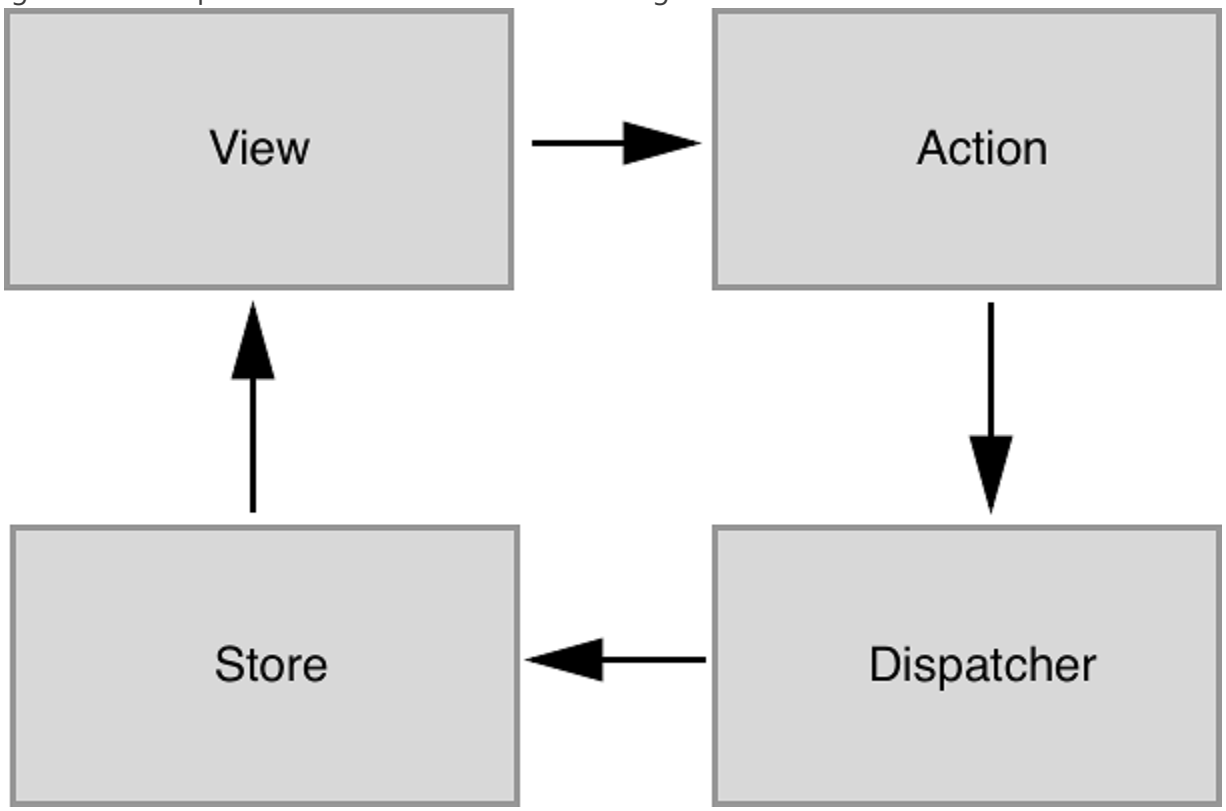
Flux is foremost an architecture pattern. It was developed as an alternative to the prevailing Model-View-Controller (MVC) JavaScript patterns popularized by incumbent frameworks, such as Backbone, Angular, or Ember.

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 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 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.

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.

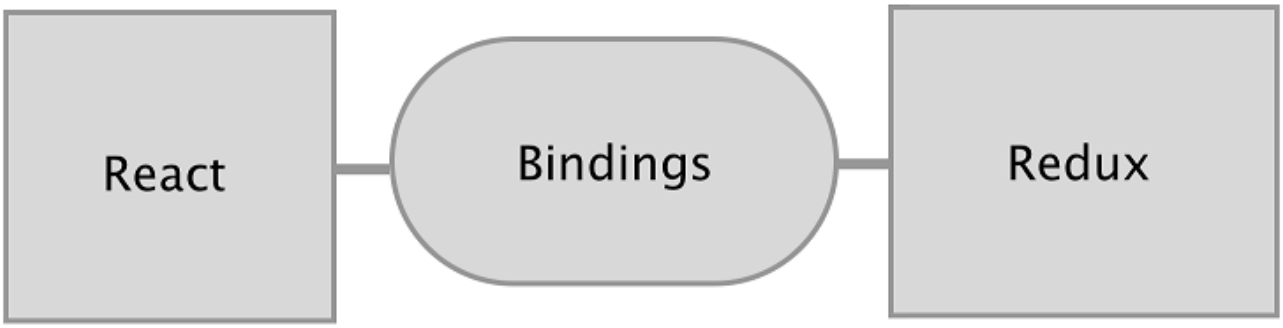
Redux enjoys the benefits of a predictable data flow from the Flux architecture, but also found ways to alleviate the uncertainty of store callback registrations. As alluded to in the previous section, it can be a pain to reconcile the state of multiple Flux stores. Redux,

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.



It turns out that the Redux paradigm for state management can be implemented alongside most JavaScript frameworks. Bindings exist for Angular 1 and 5, Backbone, Ember, and many more technologies.

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 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.

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

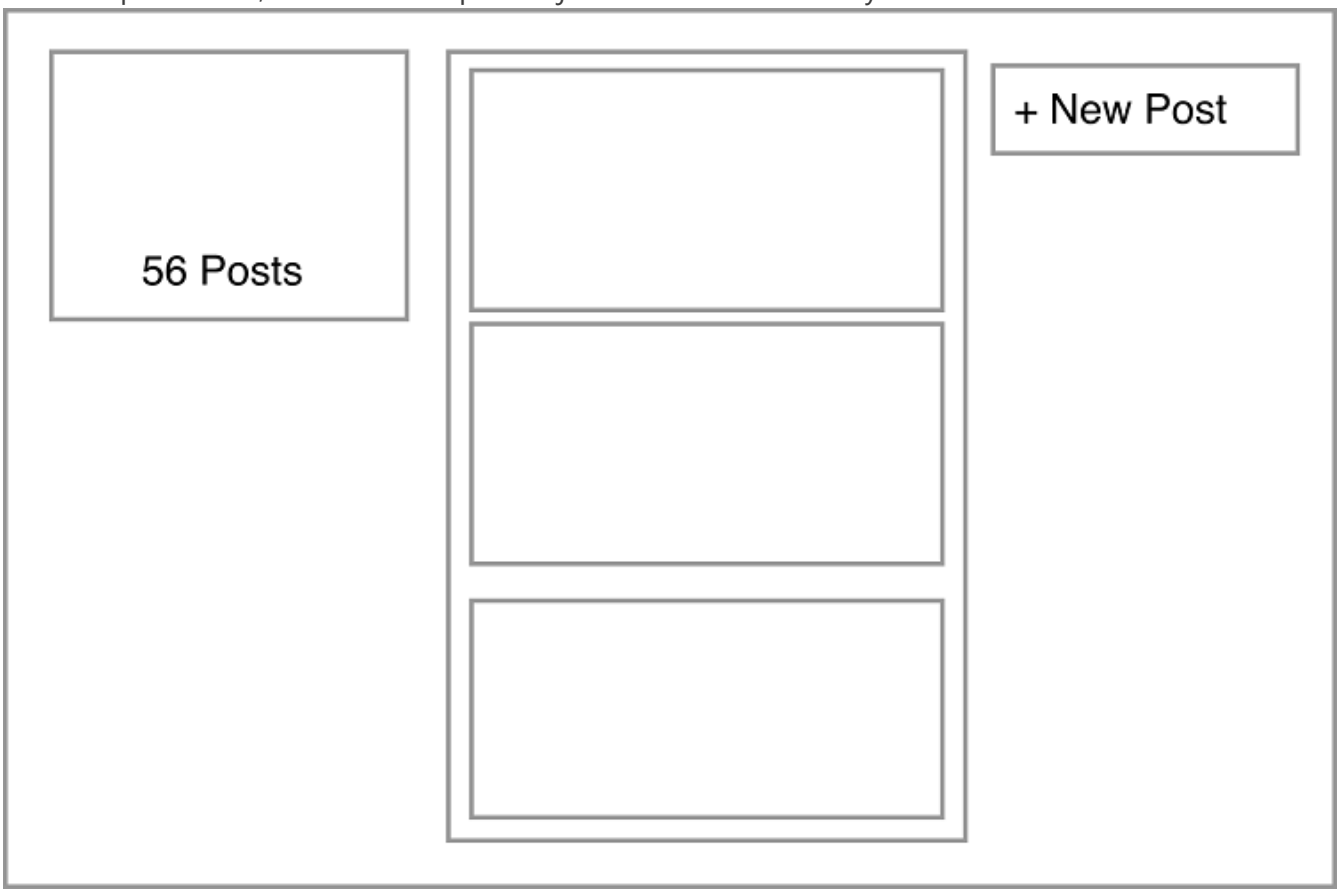
So far, we've touch briefly upon things like actions, reducers, and the store, but in this section, we'll cover each in more depth. What's important to take away here is the role that each element plays, and how they work together to produce a desired result.

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.

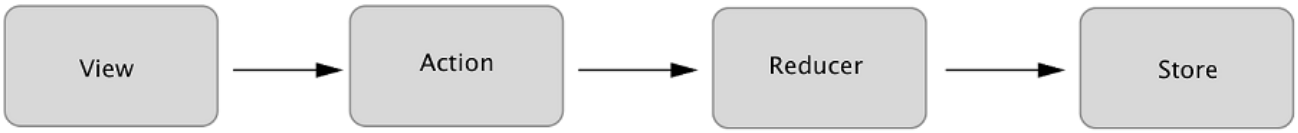


The following distinct steps are involved in handling an event like a new post:

- 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

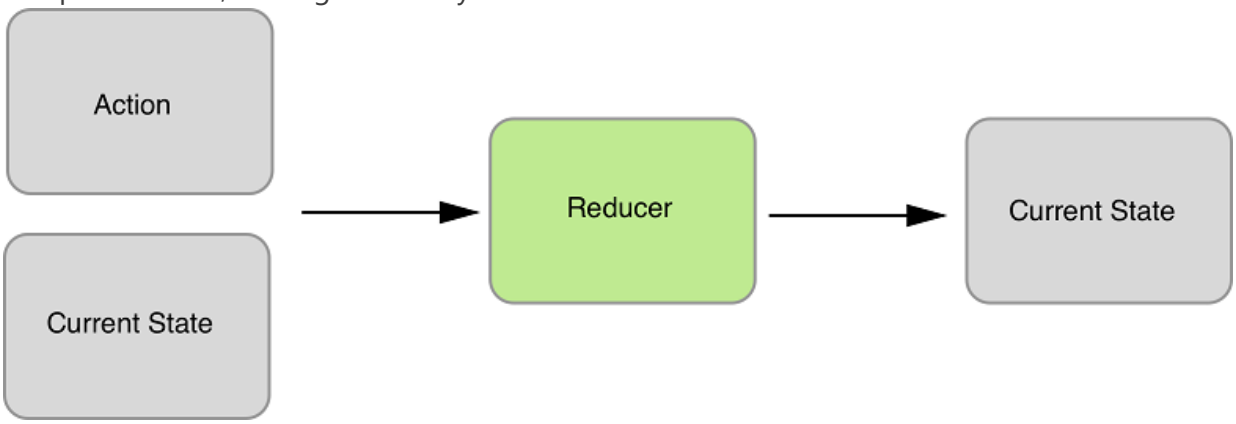
We want to do two things in response to the user submitting a new post: add the post to the user's activity feed and increment their total post count. After the user submits, we'll kick off the process by dispatching an action. Actions are plain old JavaScript objects that represent an event in your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

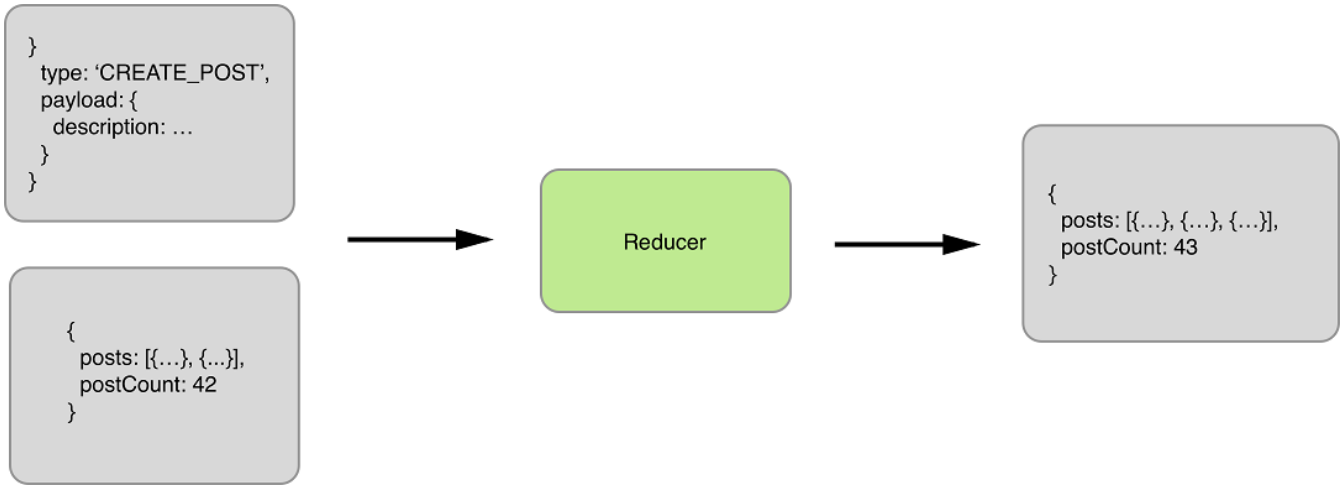
- type — a string that represents the category of action being performed. By convention, this property is capitalized and uses underscores as delimiters.
- payload — an object that provides the data necessary to perform the action. In our case, we only need one field: the contents of the message we want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

- 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:

-
- 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.



- 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.

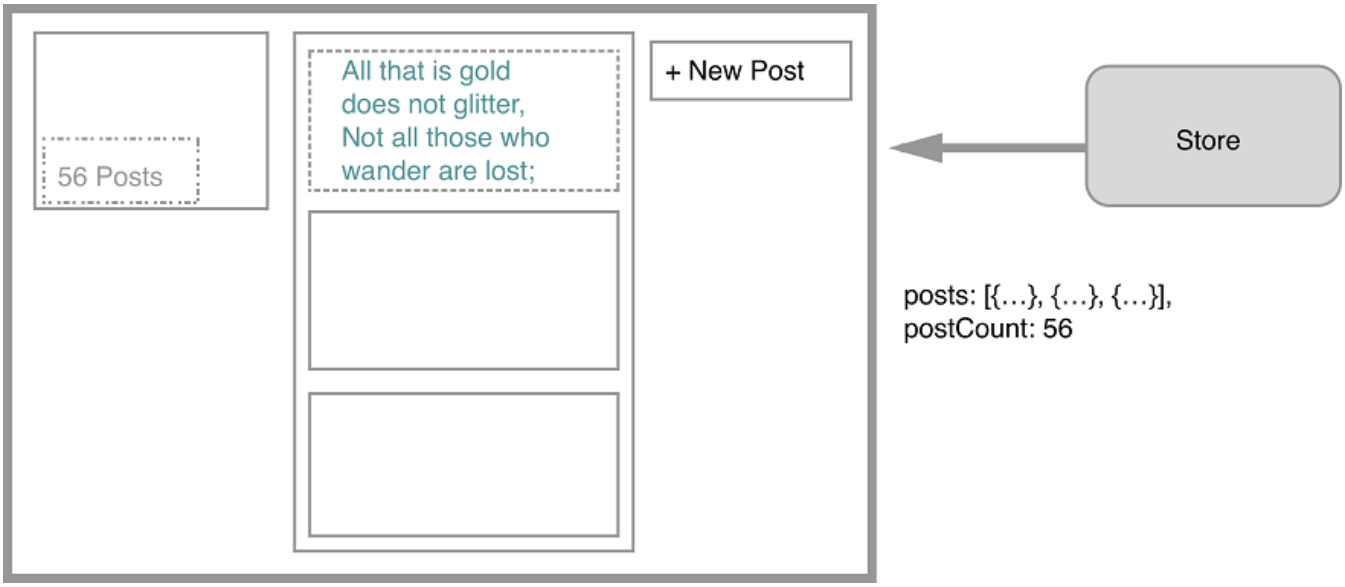
In Redux, application state is stored in a single object. The store has a few main roles, which follow:

- 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.

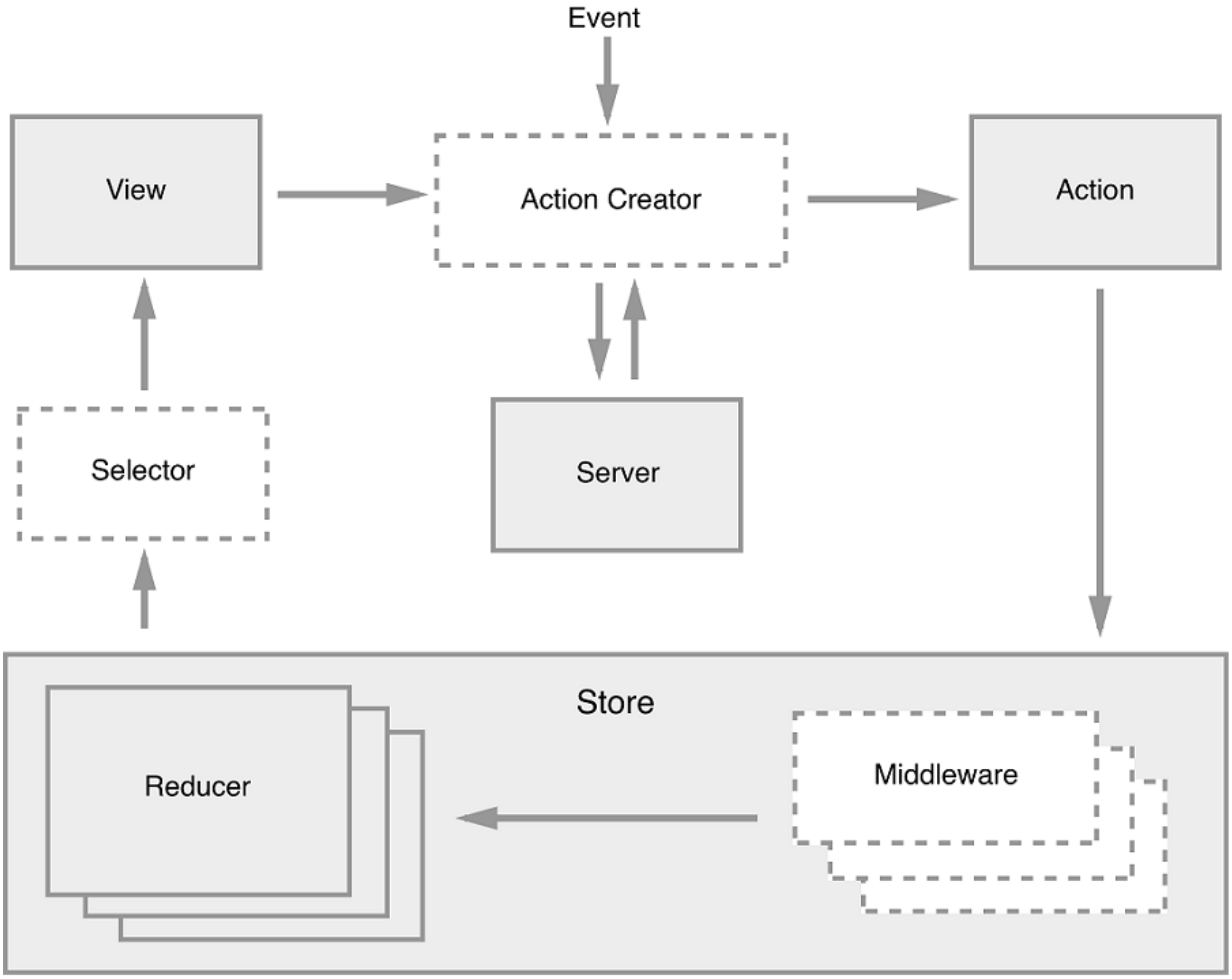
The reducer processed the action and computed the next state. Now it's time for the store to update itself and broadcast the new state to all registered listeners (we care specifically about the components that make up our profile page).

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.

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.

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 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 a lot of overlap in usage between the packages.

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.

Working with React & Redux

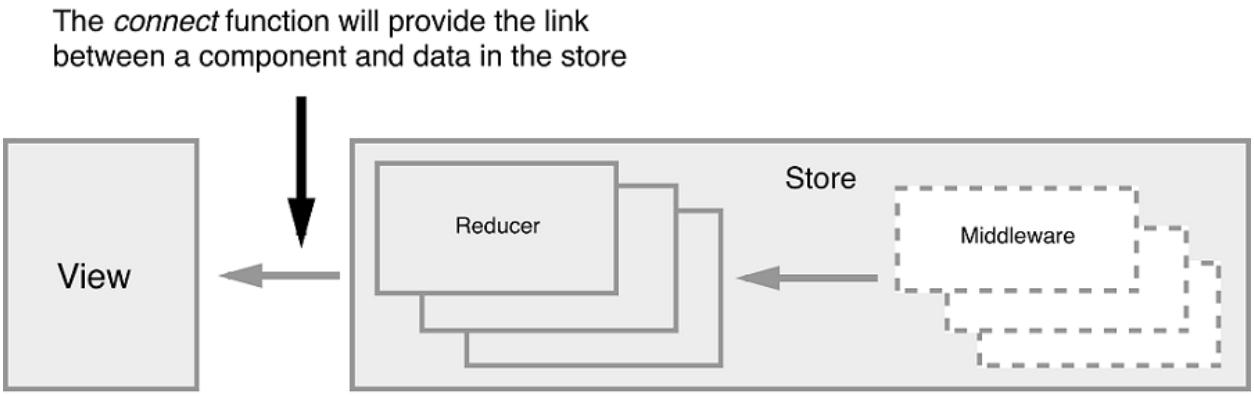
state — is held in a container known as the store.

There can only be one of these within an application.

The store is essentially a state tree where states for all objects are kept. Any UI component can access the state of a particular object directly from the store.

To change a state from a local or remote component, an action needs to be dispatched.

Dispatch in this context means sending actionable information to the store. When a store receives an action, it delegates it to the relevant reducer. A reducer is simply a pure function that looks at the previous state, performs an action and returns a new state.



Understand Immutability First

Before we start, I need you to first understand what **immutability** means in JavaScript. According to the Oxford English Dictionary, immutability means being **unchangeable**. In programming, we write code that changes the values of variables all the time. This is referred to as **mutability**.

The way we do this can often cause unexpected bugs in our projects. If your code only deals with primitive data types (numbers, strings, booleans), then you don't need to worry. However, if you're working with Arrays and Objects, performing **mutable** operations on them can create unexpected bugs.

let's create an array, then later assign it to another variable:

```
> let a = [1,2,3]
> let b = a
> b.push(9)
> console.log(b)
[ 1, 2, 3, 9 ] // b output
> console.log(a)
[ 1, 2, 3, 9 ] // a output
```

As you can see, updating array b caused array a to change as well. This happens because Objects and Arrays are known **referential data types** — meaning that such data types don't actually hold values themselves, but are pointers to a memory location where the values are stored. By assigning a to b, we merely created a second pointer that references the same location. To fix this, we need to copy the referenced values to a new location. In JavaScript, there are three different ways of achieving this:

1. using immutable data structures created by Immutable.js
2. using JavaScript libraries such as Underscore and Lodash to execute immutable operations
3. using native **ES6** functions to execute immutable operations.

For this article, we'll use the **ES6** way, since it's already available in the NodeJS environment. Inside your NodeJS terminal, execute the following:

```
> a = [1,2,3] // reset a
[ 1, 2, 3 ]
> b = Object.assign([],a) // copy array a to b
[ 1, 2, 3 ]
> b.push(8)
> console.log(b)
[ 1, 2, 3, 8 ] // b output
> console.log(a)
[ 1, 2, 3 ] // a output
```

In the above code example, array b can now be modified without affecting array a. We've used Object.assign() to create a new copy of values that variable b will now point to. We can also use the rest operator(...) to perform an immutable operation like this:

```
> a = [1,2,3]
[ 1, 2, 3 ]
> b = [...a, 4, 5, 6]
[ 1, 2, 3, 4, 5, 6 ]
> a
[ 1, 2, 3 ]
```

The rest operator works with object literals too! I won't go deep into this subject, but here are some additional ES6 functions that we'll use to perform immutable operations:

Setting up Redux

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


const reducer = function(state, action) {
  return state;
}
```

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

Let me explain what the above piece of code does:

- **1st statement**. We import a createStore() function from the Redux package.
- **2nd statement**. We create an empty function known as a **reducer**. The first argument, state, is current data held in the store. The second argument, action, is a container for:
  - **type** — a simple string constant e.g. ADD, UPDATE, DELETE etc.
  - **payload** — data for updating state
- **3rd statement**. We create a Redux store, which can only be constructed using a reducer as a parameter. The data kept in the Redux store can be accessed directly, but can only be updated via the supplied reducer.

You may have noticed I mentioned current data as if it already exists. Currently, our state is undefined or null. To remedy this, just assign a default value to state like this to make it an empty array:

```
const reducer = function(state=[], action) {
  return state;
}
```

Now, let's get practical. The reducer we created is generic. Its name doesn't describe what it's for. Then there's the issue of how we work with multiple reducers. The answer is to use a combineReducers function that's supplied by the Redux package. Update your code as follows:

```
// src/index.js
…
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {
  return state;
}

const cartReducer = function(state=[], action) {
  return state;
}

const allReducers = {
```

```
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);
```

In the code above, we've renamed the generic reducer to cartReducer. There's also a new empty reducer named productsReducer that I've created just to show you how to combine multiple reducers within a single store using the combineReducersfunction.

Next, we'll look at how we can define some test data for our reducers. Update the code as follows:

```
// src/index.js
…
const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

const cartReducer = function(state=initialState, action) {
  return state;
}
```

```
…
let store = createStore(rootReducer);


console.log("initial state: ", store.getState());
```

Just to confirm that the store has some initial data, we use store.getState() to print out the current state in the console. You can run the dev server by executing npm start or yarn start in the console. Then press Ctrl+Shift+I to open the inspector tab in Chrome in order to view the console tab.

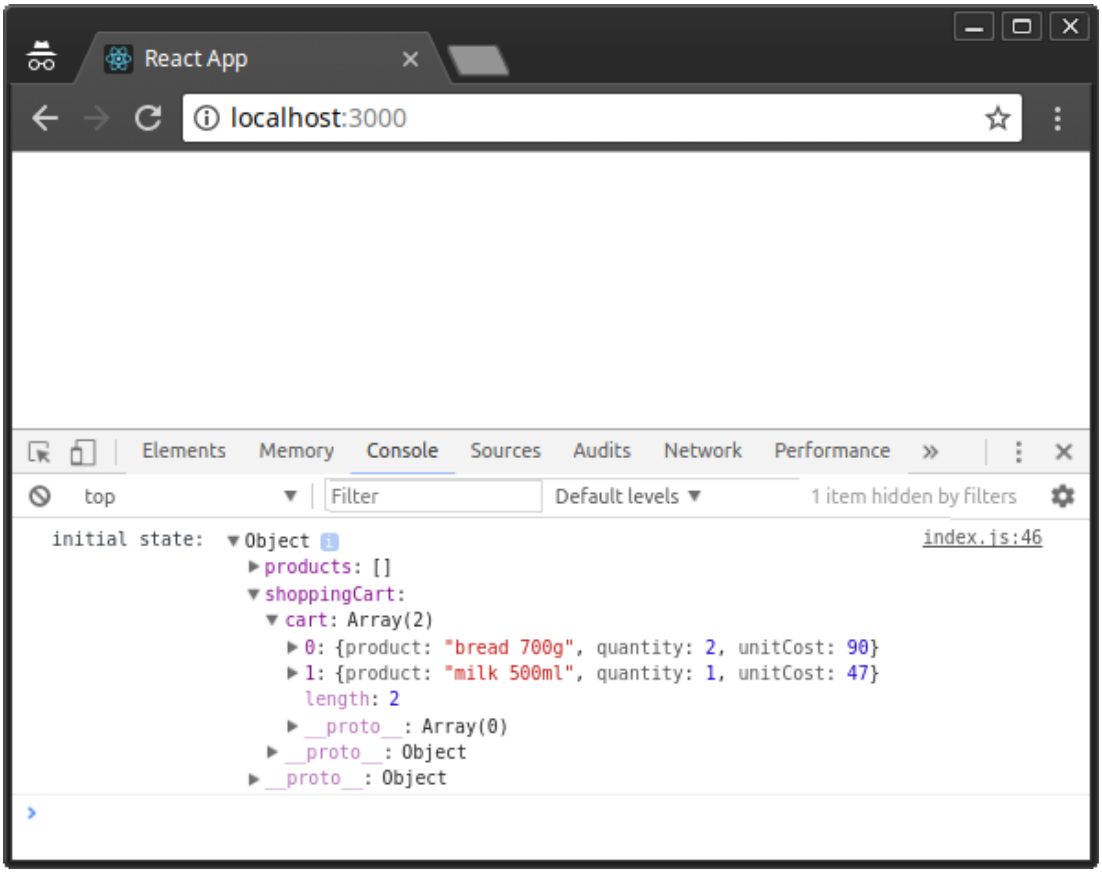


Figure 2: Redux Initial State

Currently, our cartReducer does nothing, yet it's supposed to manage the state of our shopping cart items within the Redux store. We need to define actions for adding, updating and deleting shopping cart items. Let's start by defining logic for a ADD_TO_CART action:

```
// src/index.js
…
const ADD_TO_CART = 'ADD_TO_CART';


const cartReducer = function(state=initialState, action) {
```

```
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}
…
```

Take your time to analyze and understand the code. A reducer is expected to handle different action types, hence the need for a SWITCH statement. When an action of type ADD_TO_CART is dispatched anywhere in the application, the code defined here will handle it. As you can see, we're using the information provided in action.payload to combine to an existing state in order to create a new state.

Next, we'll define an action, which is needed as a parameter for store.dispatch(). **Actions** are simply JavaScript objects that must have type and an optional payload. Let's go ahead and define one right after the cartReducer function:

```
…
function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
…
```

Here, we've defined a function that returns a plain JavaScript object. Nothing fancy. Before we dispatch, let's add some code that will allow us to listen to store event changes. Place this code right after the console.log() statement:

```
…
```

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

unsubscribe();
```

Next, let's add several items to the cart by dispatching actions to the store. Place this code before unsubscribe():

```
...
store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));
```

For clarification purposes, I'll illustrate below how the entire code should look after making all the above changes:

```
// src/index.js

import { createStore } from "redux";
import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {
  return state;
}

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
    {
      product: 'milk 500ml',
      quantity: 1,
```

```
      unitCost: 47
    }
  ]
}

const ADD_TO_CART = 'ADD_TO_CART';

const cartReducer = function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}

function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: {
      product,
      quantity,
      unitCost
    }
  }
}
```

```
const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);

console.log("initial state: ", store.getState());

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

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've saved your code, Chrome should automatically refresh. Check the console tab to confirm that the new items have been added:

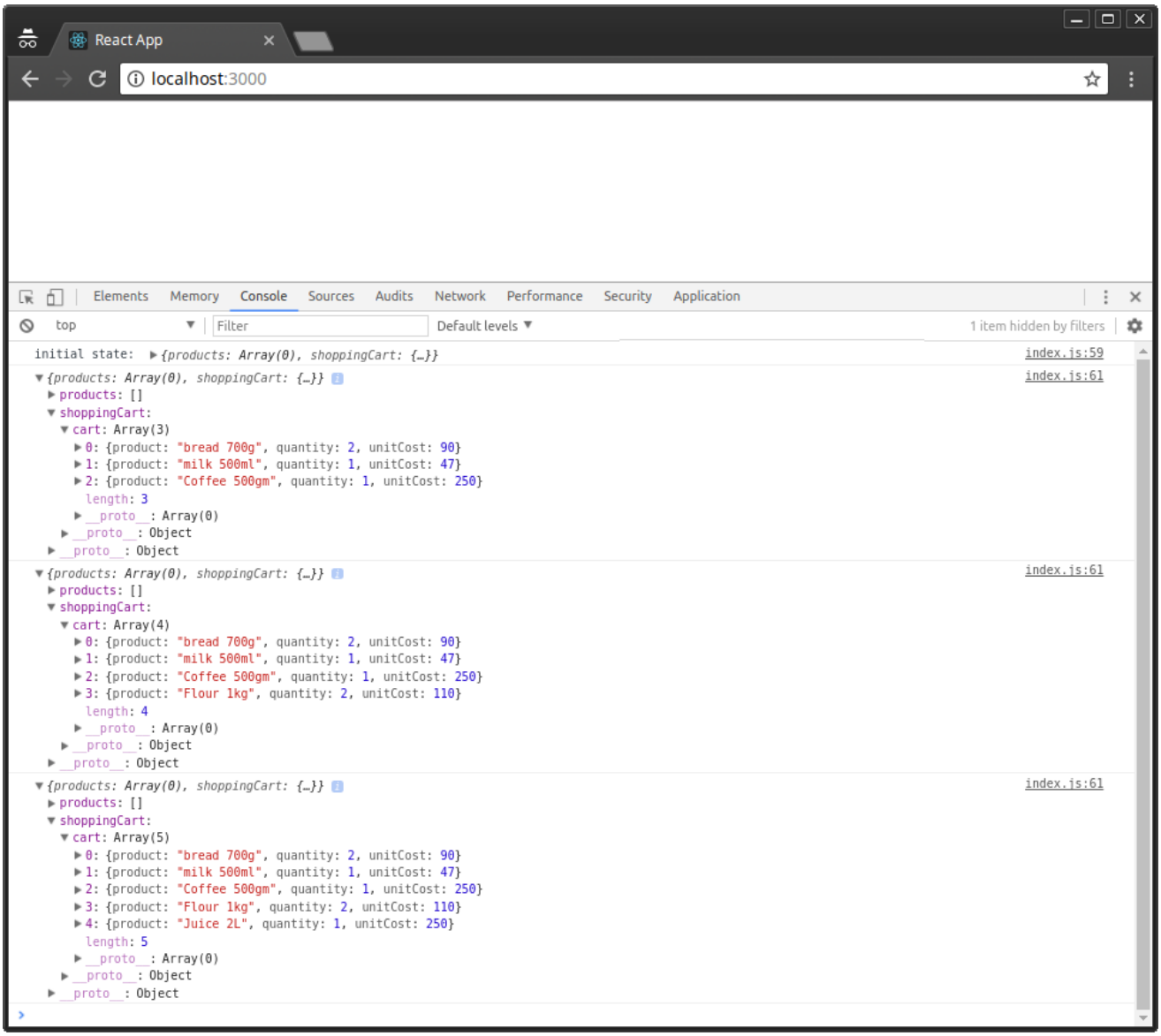


Figure 3: Redux Actions Dispatched

Organizing Redux Code

The index.js file has quickly grown large. This is not how Redux code is written. I've only done this to show you how simple Redux is. Let's look at how a Redux project should be organized. First, create the following folders and files within the src folder, as illustrated below:

```
src/
├── actions
│   └── cart-actions.js
├── index.js
```

```
├── reducers
│   ├── cart-reducer.js
│   ├── index.js
│   └── products-reducer.js
└── store.js
```

Next, let's start moving code from index.js to the relevant files:

```
// src/actions/cart-actions.js

export const ADD_TO_CART = 'ADD_TO_CART';

export function addToCart(product, quantity, unitCost) {
  return {
    type: ADD_TO_CART,
    payload: { product, quantity, unitCost }
  }
}
// src/reducers/products-reducer.js

export default function(state=[], action) {
  return state;
}
// src/reducers/cart-reducer.js

import  { ADD_TO_CART }  from '../actions/cart-actions';

const initialState = {
  cart: [
    {
      product: 'bread 700g',
      quantity: 2,
      unitCost: 90
    },
```

```
    {
      product: 'milk 500ml',
      quantity: 1,
      unitCost: 47
    }
  ]
}

export default function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
    }

    default:
      return state;
  }
}
// src/reducers/index.js

import { combineReducers } from 'redux';
import productsReducer from './products-reducer';
import cartReducer from './cart-reducer';

const allReducers = {
  products: productsReducer,
  shoppingCart: cartReducer
}
```

```
const rootReducer = combineReducers(allReducers);

export default rootReducer;
// src/store.js

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

let store = createStore(rootReducer);

export default store;
// src/index.js

import store from './store.js';
import { addToCart }  from './actions/cart-actions';

console.log("initial state: ", store.getState());

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

store.dispatch(addToCart('Coffee 500gm', 1, 250));
store.dispatch(addToCart('Flour 1kg', 2, 110));
store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();
```

After you've finished updating the code, the application should run as before now that it's better organized. Let's now look at how we can update and delete items from the shopping cart. Open cart-reducer.js and update the code as follows:

```
// src/reducers/cart-actions.js
…
```

```
export const UPDATE_CART = 'UPDATE_CART';
export const DELETE_FROM_CART = 'DELETE_FROM_CART';
…
export function updateCart(product, quantity, unitCost) {
  return {
    type: UPDATE_CART,
    payload: {
      product,
      quantity,
      unitCost
    }
  }
}

export function deleteFromCart(product) {
  return {
    type: DELETE_FROM_CART,
    payload: {
      product
    }
  }
}
```

Next, update cart-reducer.js as follows:

```
// src/reducers/cart-reducer.js
…
export default function(state=initialState, action) {
  switch (action.type) {
    case ADD_TO_CART: {
      return {
        ...state,
        cart: [...state.cart, action.payload]
      }
```

```
    }

    case UPDATE_CART: {
      return {
        ...state,
        cart: state.cart.map(item => item.product ===
action.payload.product ? action.payload : item)
      }
    }

    case DELETE_FROM_CART: {
      return {
        ...state,
        cart: state.cart.filter(item => item.product !==
action.payload.product)
      }
    }

    default:
      return state;
  }
}
```

Finally, let's dispatch the UPDATE_CART and DELETE_FROM_CART actions in index.js:

```
// src/index.js
…
// Update Cart
store.dispatch(updateCart('Flour 1kg', 5, 110));

// Delete from Cart
store.dispatch(deleteFromCart('Coffee 500gm'));
…
```

Your browser should automatically refresh once you've saved all the changes. Check the console tab to confirm the results:

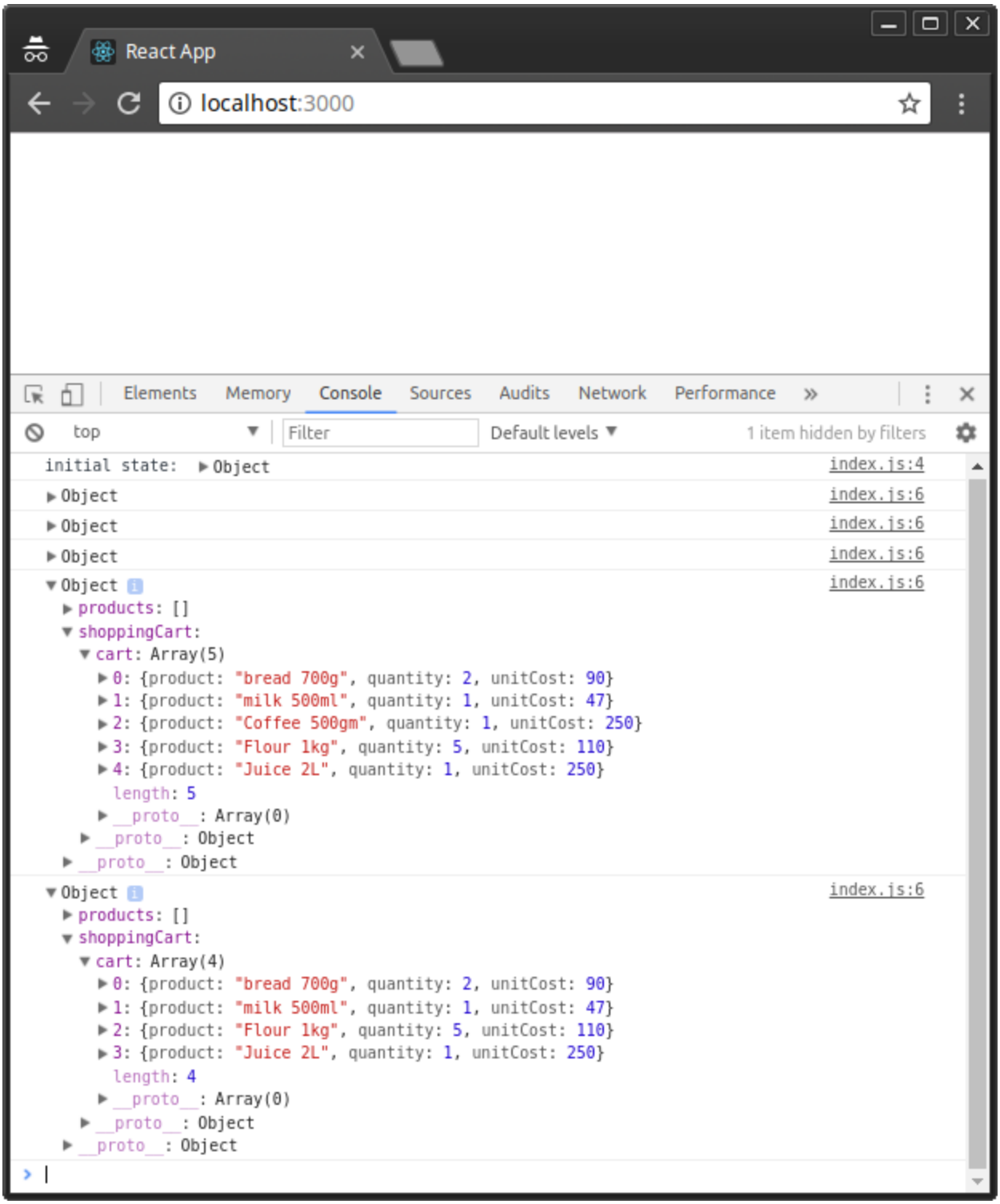


Figure 4: Redux Update and Delete Actions

As confirmed, the quantity for 1kg of flour is updated from 2 to 5, while the the 500gm of coffee gets deleted from cart.

Debugging with Redux tools

Now, if we've made a mistake in our code, how do we debug a Redux project?

Redux comes with a lot of third-party debugging tools we can use to analyze code behavior and fix bugs. Probably the most popular one is the **time-travelling tool**, otherwise known as redux-devtools-extension.

Setting it up is a 3-step process. First, go to your Chrome browser and install the Redux Devtools extension.

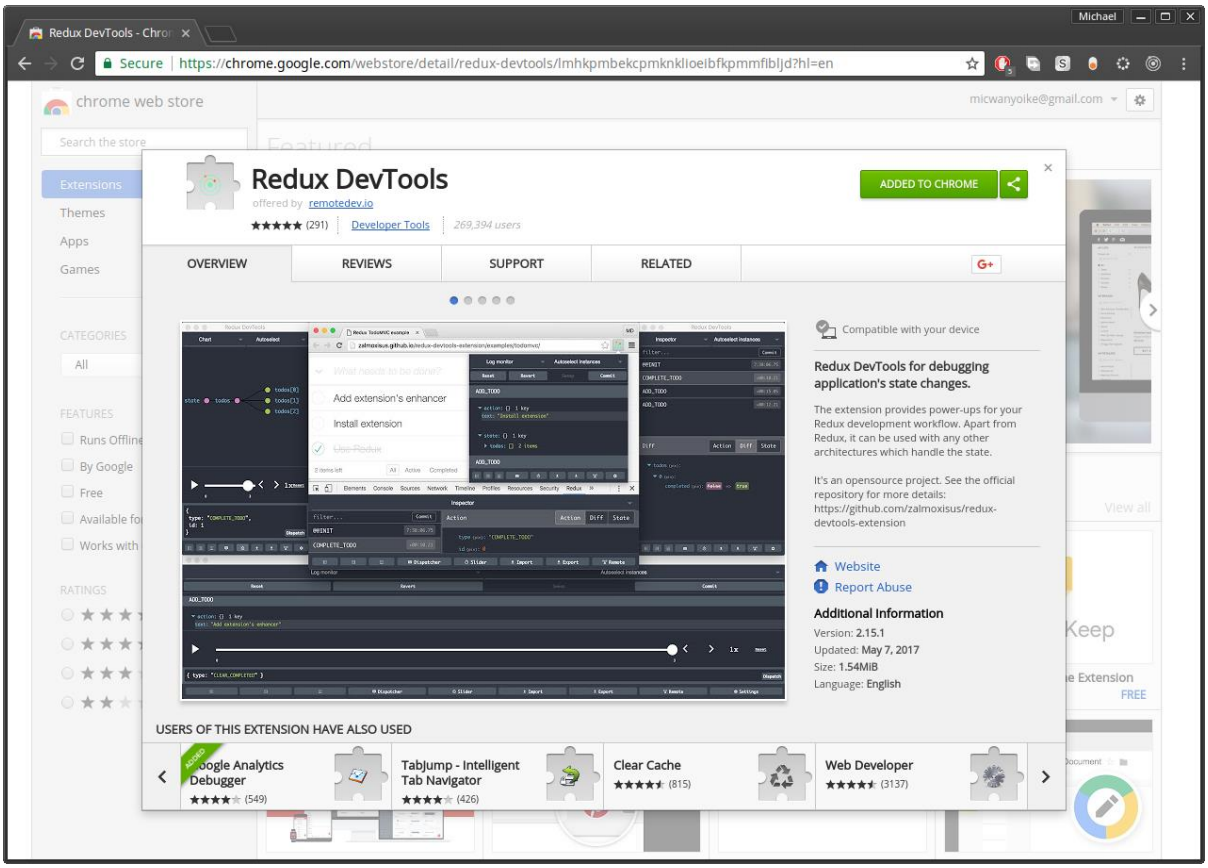


Figure 5: Redux DevTools Chrome Extensions

Next, go to your terminal where your Redux application is running and press Ctrl+Cto stop the development server. Next, use npm or yarn to install the redux-devtools-extension package. Personally, I prefer Yarn, since there's a yarn.lock file that I'd like to keep updated.

```
yarn add redux-devtools-extension
```

Once installation is complete, you can start the development server as we implement the final step of implementing the tool. Open store.js and replace the existing code as follows:

```
// src/store.js
import { createStore } from "redux";
import { composeWithDevTools } from 'redux-devtools-extension';
import rootReducer from './reducers';

const store = createStore(rootReducer, composeWithDevTools());

export default store;
```

Integration with React

At the beginning of this tutorial, I mentioned Redux really pairs well with React. Well, you only need a few steps to setup the integration. Firstly, stop the development server, as we'll need to install the react-redux package, the official Redux bindings for React:

```
yarn add react-redux
```

Next, update index.js to include some React code. We'll also use the Providerclass to wrap the React application within the Redux container:

```
// src/index.js
…
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';

const App = <h1>Redux Shopping Cart</h1>;

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

```
);
...
```

Just like that, we've completed the first part of the integration.