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 3

brief contents

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 only 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, you 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 part 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 look at what the developer tools might look like in the context of Parsnip (figure 3.1), the task management application you started in chapter 2.

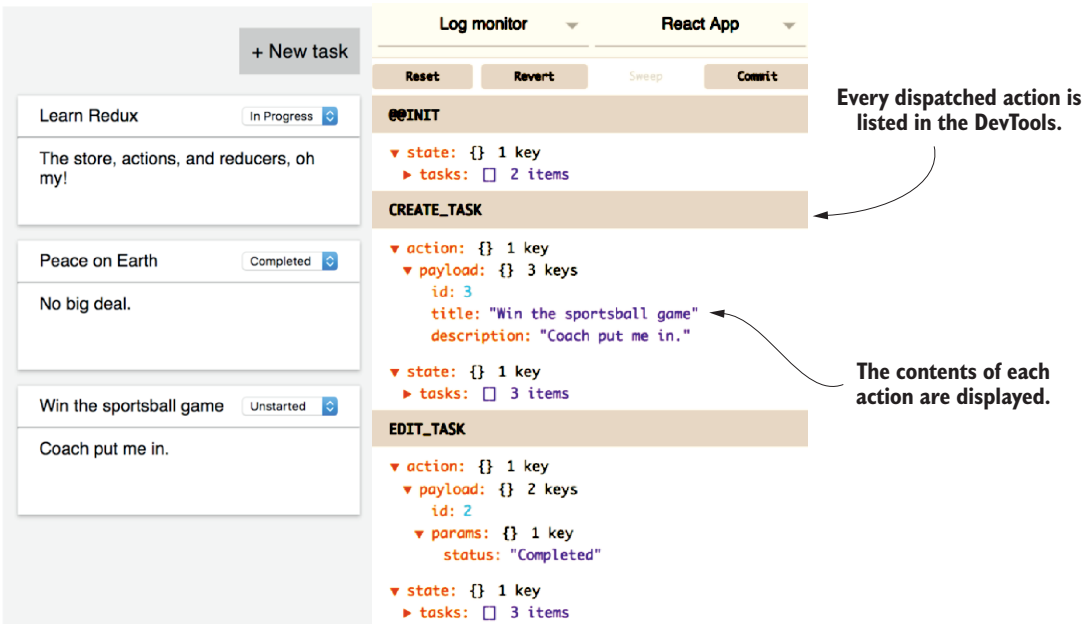

Figure 3.1 The Redux DevTools can be used to display actions in real time.

In the right panel (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 you've been up to within Parsnip without having to observe you 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 it's dispatched. For every action, you can be sure that the payload looks the way you intended. Still, there's much more that you can do with the DevTools.

You'll notice that beneath each action in the right page is a snapshot of the Redux store. Not only can you view the action produced, you can see the application state

that resulted from that action. Better still, you can dig into the Redux store and see, highlighted, the exact values that changed as a result of the action. Figure 3.2 points out these details within the DevTools.

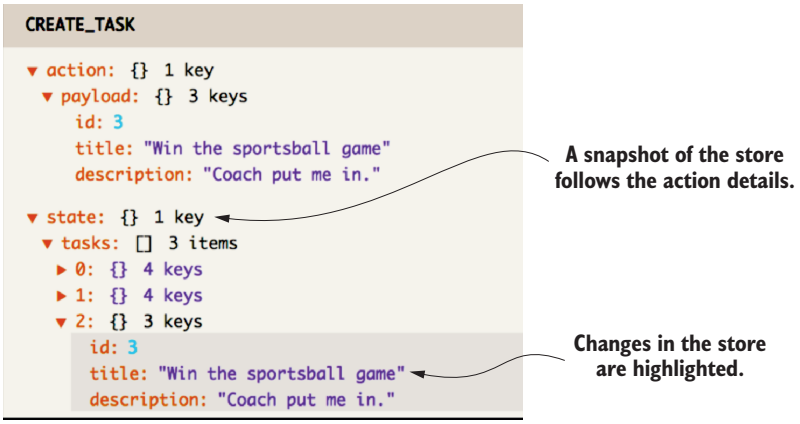

Figure 3.2 The DevTools highlight attributes of the Redux store that have changed as a result of the action.

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 a long list of actions. Click it again, and the application returns to its original state. See figure 3.3 for an example of this toggled state.

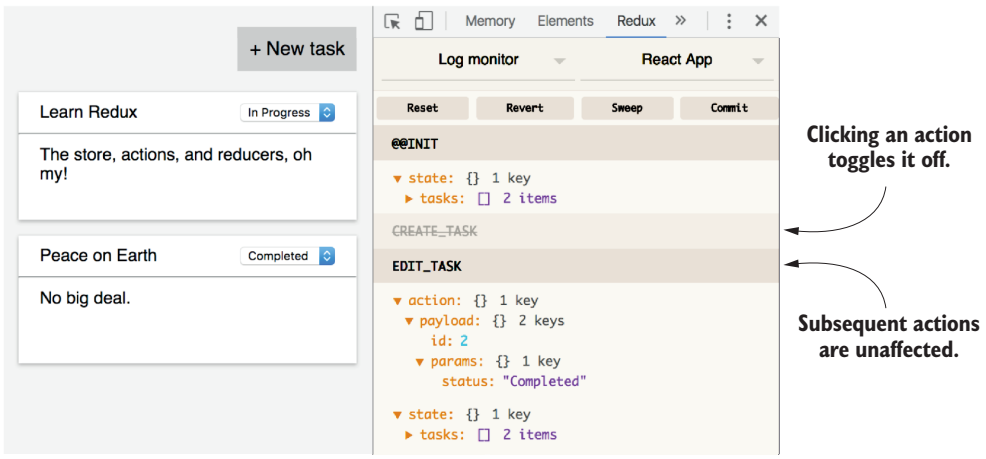

Figure 3.3 Toggling an action recalculates the application state as if the action was never dispatched.

The rewinding and replaying of actions like this is what inspired the name time-travel debugging. To determine what Parsnip would look like if you hadn't created a third task, there's no need to refresh and re-create the state—you can 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's a feature of the DevTools. More specifically, it's a feature of many DevTools monitors.

3.3 Visualizing changes with DevTools monitors

The Redux DevTools provide an interface to the actions and Redux store, but don't 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.

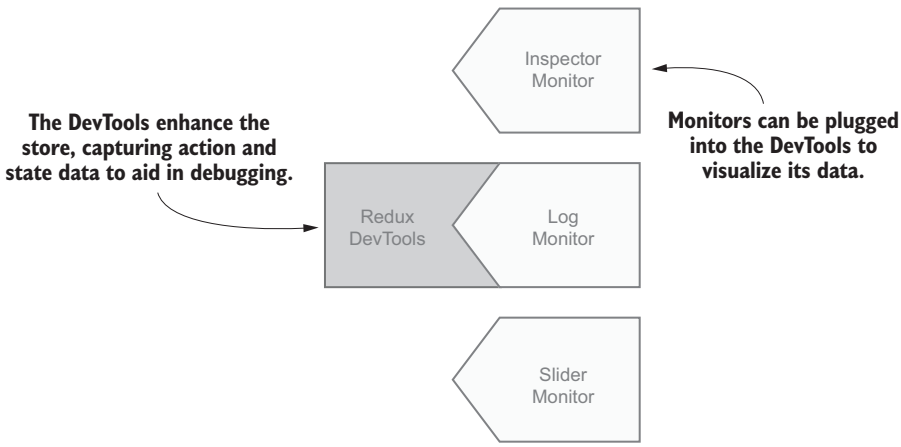

Figure 3.4 Various monitors can be combined with the Redux DevTools to visualize actions and store data.

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 viewed a simple 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 storing of 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're 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. Here 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 using 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 your needs out of the box.

As JavaScript developers, many of us already spend much 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 adds one more panel, and clicking it reveals the Inspector and other monitors. You 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.

This process has two steps: installing the Chrome browser extension and hooking the Redux DevTools into the store. Installing the Chrome extension is the simpler of the two. Visit the Chrome Web Store, search for Redux DevTools, and install the first package by the author remotedev.io.

Now 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. You'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, you'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, Redux presumes the second argument is an enhancer and there's no initial state. The following listing is an example of this case. Enhancers are a way to augment the Redux store and the `devToolsEnhancer` function is doing 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, you can find the Redux DevTools by selecting the new Redux panel, made available by the Redux DevTools Chrome extension (figure 3.5).

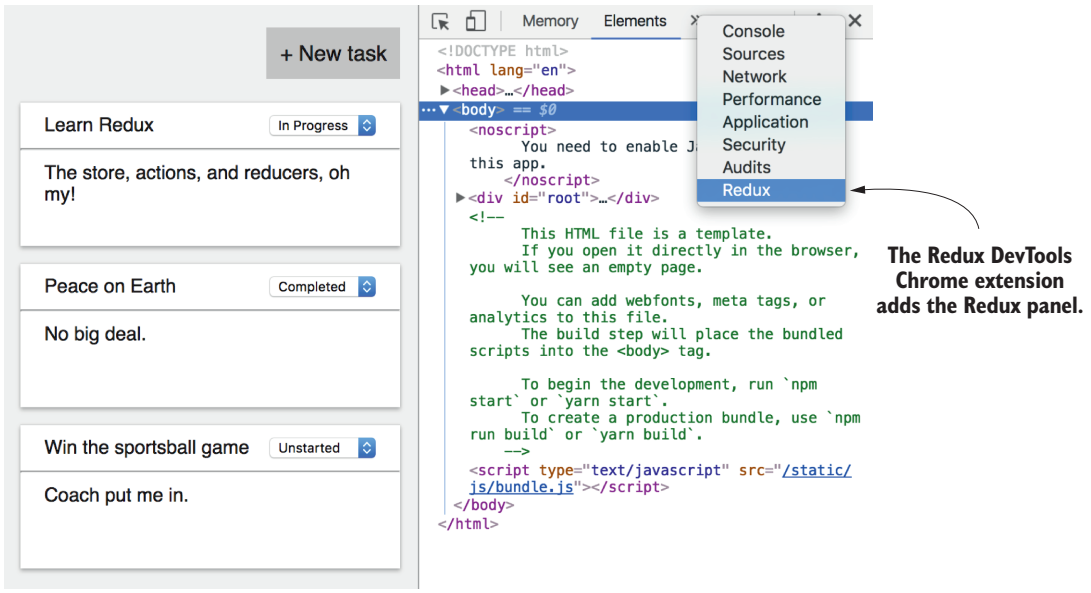

Figure 3.5 Navigate to the Redux DevTools via the Redux panel in the Chrome developer tools.

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 by adding a couple of 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 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 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.

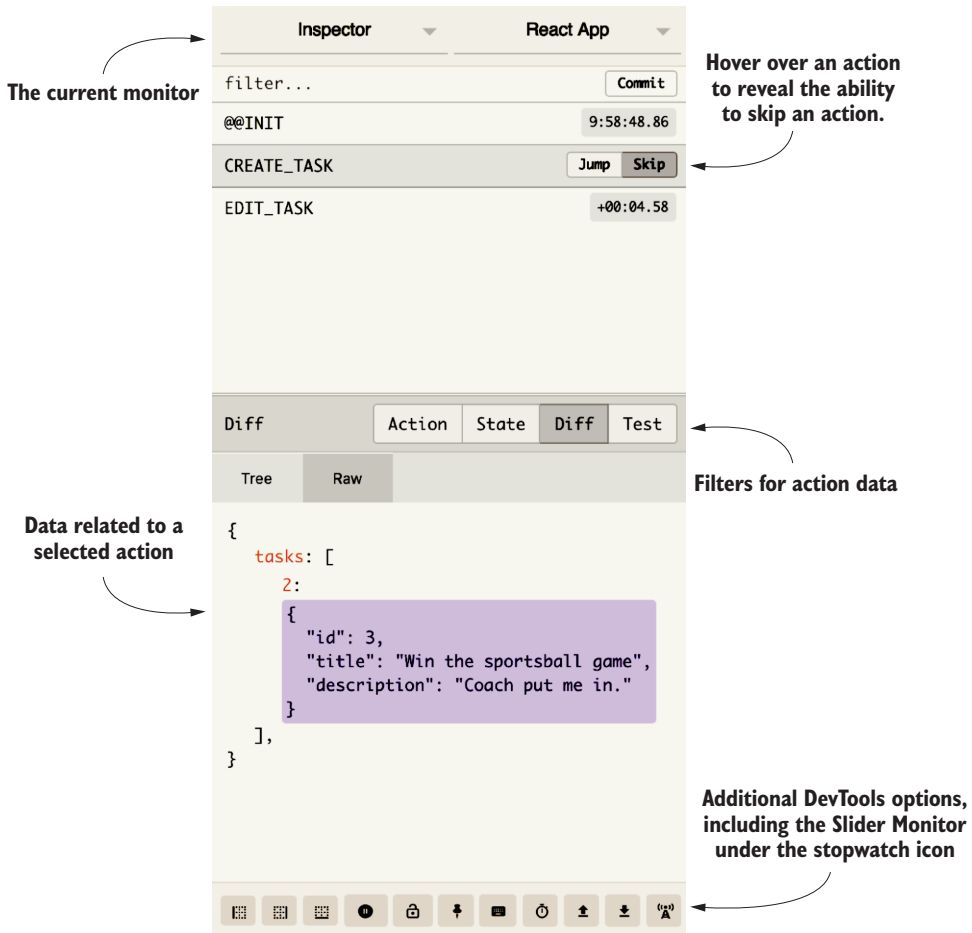

Figure 3.6 The skip, Diff filter, and monitor menu options offer unique ways to visualize and debug state effects.

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 using console logs or alerts, this developer experience leaves much 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 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 you no longer had to manually refresh the page after a change is made to your code? If you know you need to refresh the browser to view and test each code change, your build tools might as well take care of that for you.

Multiple tools are capable of this file-watching and updating on change, but we'll reference Webpack specifically throughout the rest of this chapter. Webpack isn't required to use Redux, but it's 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, you're interested in only those features that improve your Redux development workflow. Don't panic, no Webpack expertise is required to understand the concepts in this chapter.

Webpack can save you 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 you 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 incremental improvements in a development workflow. You may be tempted to believe the Webpack bang isn't worth the buck, but you'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 isn't 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. You'll want to configure two specific updates 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 you 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`. The `accept` command takes two arguments: one or more dependencies and a callback. You'll want an update to any of your components to trigger the hot replacement, and fortunately, you don't have to list every React component in your application as a dependency. Whenever a child of the top-most component updates, the change will be picked up by the parent. You can 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 your case, you want 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, and then those changes are rendered to the DOM without reloading the page.

Listing 3.2 src/index.js

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

The Create React App has hot module replacement enabled in development mode.

Whenever the App component (or one of its children) changes, re-render the component.

Webpack won't expose `module.hot` in a production environment for good reason: you 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 its respective components update in real time.

In listing 3.3, you see a similar pattern to the implementation for components. In development mode, you listen for changes to your 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, you'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', () => {
      const nextRootReducer = require('./reducers').default;
      store.replaceReducer(nextRootReducer);
   });
}
```

Whenever the reducer updates, perform the hot module replacement.

The Redux store has a replaceReducer method to facilitate this update.

Imagine that as you're developing the workflow for the `CREATE_TASK` action, you misspelled `CREATE_TASK` in the reducer. You might've created several tasks while testing the code you wrote, and even seen the actions logged in the Redux DevTools, but no new tasks appeared in the UI. With hot module replacement applied to the reducer, the correction of that typo results 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. You must rely on the reducer function to be deterministic; the same action will always produce the same change in state. If the Redux store is mutable, you can't be certain that a list of actions results 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 recreate the same state manually.

Now you're cooking with fire! When developing, you 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 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 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.

One tool 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 configuration of either Babel or Webpack, which necessitates ejecting (`npm run eject`) from Create React App. We won't eject from the application in this book, so implementing React Hot Loader is an exercise for you to do later. 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 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 the name of the current monitor to reveal a drop-down list of more available monitors (figure 3.7).

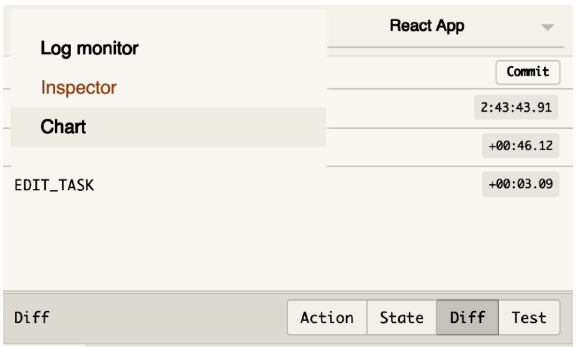

Figure 3.7 The location of the Chart Monitor

Clicking the Chart option reveals a graphical representation of the state in your application. It won't look so impressive now, because there's not much 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. See figure 3.8 for an example state payload displayed by the Chart Monitor.

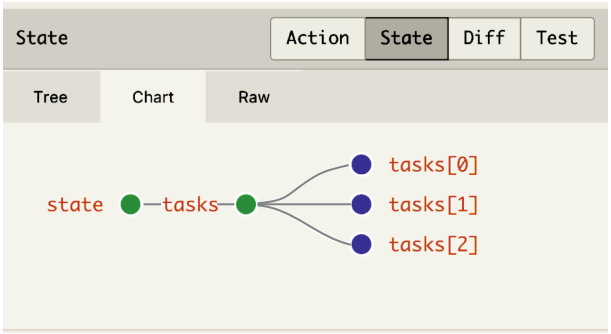

Figure 3.8 The Chart Monitor

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.

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.

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.

Summary

- 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.

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]