8 Data management in React

This chapter covers

- Introducing the importance of data management
- Evaluating various data management libraries
- Building the data layer of an application in five ways

We've been using data in all the apps we've built, from the value of an input field to the list of items in a to-do application. I've introduced several ways to handle data as well. We've used `useState` for single data values and `useReducer` for more complex multivalue state. We've also discussed how React context can help you distribute and manipulate your data throughout your application in meaningful yet simple ways.

But there are many more ways to manage data in an application. Especially as applications grow large and complex, it can make sense to use a more structured approach to data management rather than sprinkle `useState` hooks all over the codebase with no clear structure behind them.

By *data management*, I mean the process of storing, reading, and modifying data locally in the application as the user goes about their business in the application. Data in this context can be different things, including UI state (whether the menu is open or closed), form state (what is currently entered in each input), and application state (the movie object that you can edit). Data can also include read-only data, such as all the articles in the blog section of the application.

Data management is one of those topics that cause developers to turn to third-party libraries and break from using pure React. They make this change partly for historical reasons: back in the days before hooks and context, using React alone for data management was too much of a

pain. Developers also look for other libraries partly for convenience: the community has built a lot of elegant solutions to the data management problem. Different solutions are governed by different computer science principles, and if you or your team members subscribe to those same principles, those libraries will fit your project like a glove.
One data management library, Redux, is almost synonymous with React. This library was introduced in the earliest days of React and shaped the way data management worked in React for many years. It is still incredibly popular.

The downside of Redux is the amount of boilerplate code you have to write. Redux adheres to the functional coding principle of reducing state, and because it was initially strict in its application of this principle, it required a lot of moving parts to get going. If you needed Redux for a small list of items to read from, append to, and delete from, you had to create up to seven files with various configurations, action creators, stores, reducers, and more.

Redux has since become a lot more accessible in the form of Redux Toolkit (RTK), which is a much leaner library with a lower requirement for boilerplate, automating and serializing a lot of the complexity of classic Redux. Redux has also shaped how we handle data in pure React. The `useReducer` hook is inspired directly by Redux (and RTK in particular), and you'll see in this chapter that it works almost identically.

Alternatives to Redux have popped up over the years, and in this chapter, we're also going to look at one of the most bare-bones alternatives: zustand, which uses the same principles as Redux but takes the complexity all the way to zero (sacrificing scaling somewhat).

Finally, we're going to discuss a completely different approach to data management: using a state machine. A state machine is a flow management tool, but it comes with built-in data management, so it will work well for our application. The state-machine tool we'll be using is XState, and the library that connects XState to React is `@xstate/react`.

Wait—what application? Yes, we're going to build an application in this chapter so we can see how to build the data layer in five ways. But the

twist this time is that we're not going to display all the code. We want to build a larger application with about 15 components, and the source code alone would take up too many pages, so I've provided the full source code only in the online repository.
In this chapter, I will discuss only how to modify the data layer of the application and leave the presentation layer intact. We'll have a clear boundary between presentation and data management so we can replace one part without touching the other.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch08**.

8.1 Creating a goal-tracking application

All right, let's get to the application. This time, we're building a life-organizing tool—nothing less than that. A productivity and life-priority philosophy says that you should start doing something only if you can commit to doing it 100 times. If you want to start running, you should commit to running for at least 100 days. If you want to write a book, you should commit to at least 100 writing sessions. I'm not going to question or address this philosophy so much as adhere strictly to it. Thus, we'll create a tool for organizing doing something 100 times. You can do multiple things 100 times, of course.

Our inspiration comes from the aptly named website **https://do100things.com**, and in this project, we will create an application that looks somewhat like that website. I've already done that work and put it up at **https://100.bingo** for you. You can see the final product in figure 8.1.

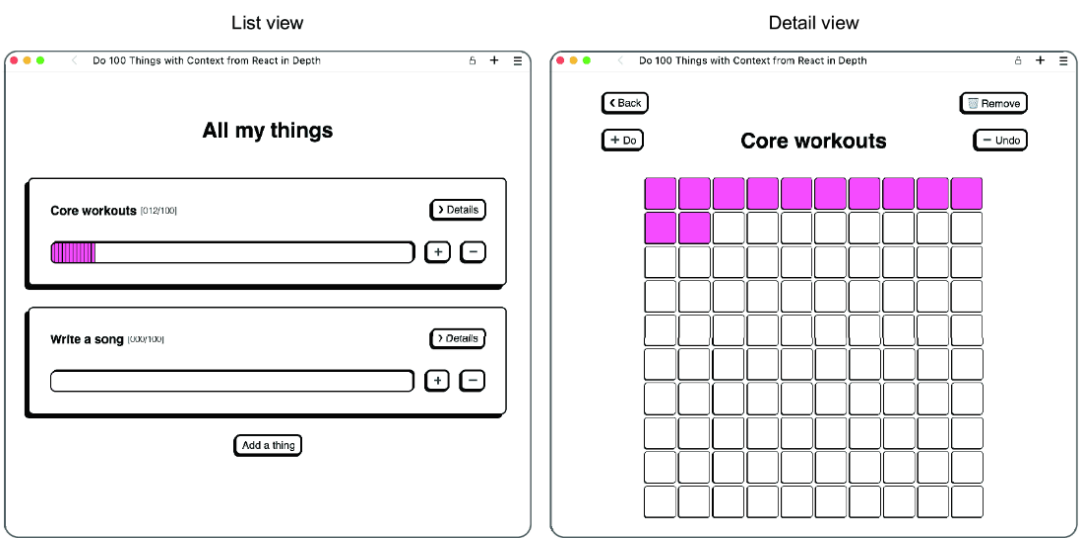

Figure 8.1 The final goal-tracking application. On the left, you see list view, where all the things are shown, and on the right, you see detail view for a single thing. We've been working on our six-packs and completed 12 sessions of core workouts. The songwriting career, however, isn't going too well.

As mentioned, I created the base application in an online repository, and we'll use all the components in this chapter. In the base repository, I created the application by using the basic `useState` hook. Using just this hook works, but it's not ideal.

In this chapter, I'll discuss how to manage this application's data by using the following methods:

- `useState` (+ context)
- `useReducer` (+ `useReduction`, `useContextSelector`, and Immer) and context
- RTK (+ Immer)
- zustand (+ Immer)
- XState (+ `xstate/react`) and context

8.2 Building the application architecture

We're building something rather simple, but because of styling (we're using styled-components), I've split it into tiny components to make getting an overview easier. From a top-down perspective, the application is structured with a presentation layer (called *things* because we're doing things in this application) and a data layer (called *data*). The application is bound together by the main `App` component, which includes the data

provider on the outside and the presentation top-level component on the inside. You can see this overall structure in figure 8.2.

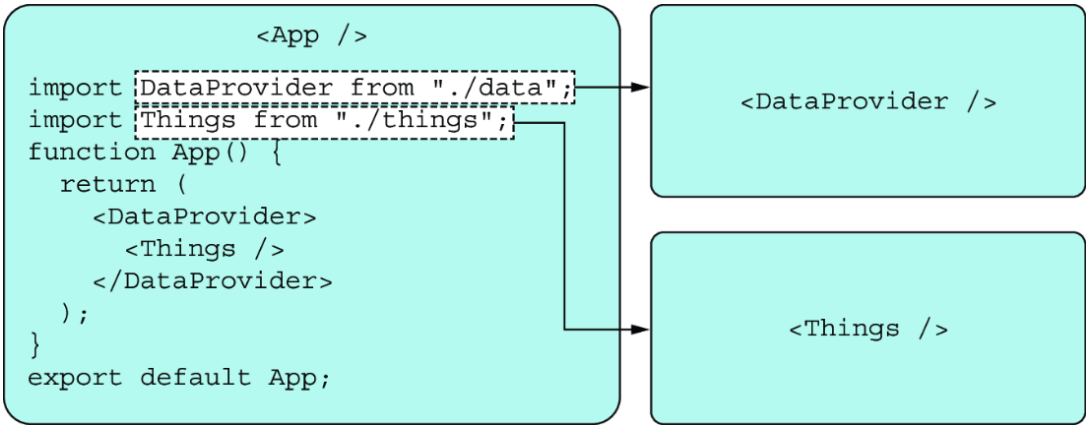

Figure 8.2 The application architecture with the main application and the two parts. The top part on the right is the data layer, and the bottom part is the presentation layer. The app on the left is merely the glue that holds the whole thing together; it doesn't do anything.

The structure of the data layer will vary with each implementation; it won't vary much, but we still have some wiggle room. The structure of the presentation layer, however, is fixed. We will use the same components with the same lines of codes in all the applications (with one notable exception in the XState variant in section 8.7.1.).

So let's see the structure of the presentation layer. First, we have two views to display: list view and single-item detail view. List view contains several things (literally) and a form for adding a new thing. Detail view has some buttons up top and a full grid of all the things you have done and must do. Figure 8.3 shows the conceptual structure.

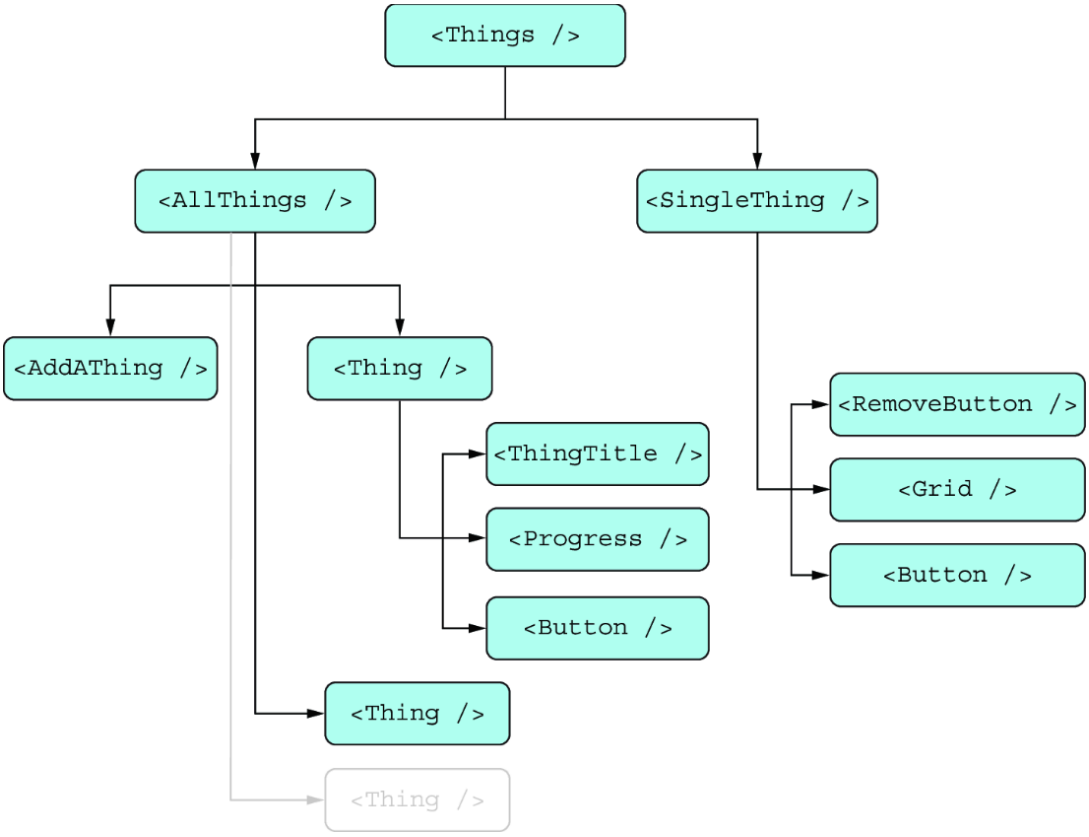

Figure 8.3 The conceptual structure of the presentation layer consists of a large number of components, with some being general purpose (such as `Button`, `Grid`, and `Progress`) and others being application specific (such as `AllThings`, `ThingTitle`, and `RemoveButton`).

The most interesting part is how these components interact with the data layer. There are four such interactions:

- The top-level `Things` component needs to know whether to display list view or detail view.
- The list view `AllThings` component needs to get an array of all the things.
- The `AddAThing` component needs to be able to add a thing to the data layer.
- The two components concerned with displaying a single thing—`Thing` and `SingleThing`—need to be able to get data for that thing and access some functions to interact with that thing.

We will create four custom hooks that expose a specific API to the presentation layer so that the relevant components can do their things. Figure 8.4 illustrates these interactions.

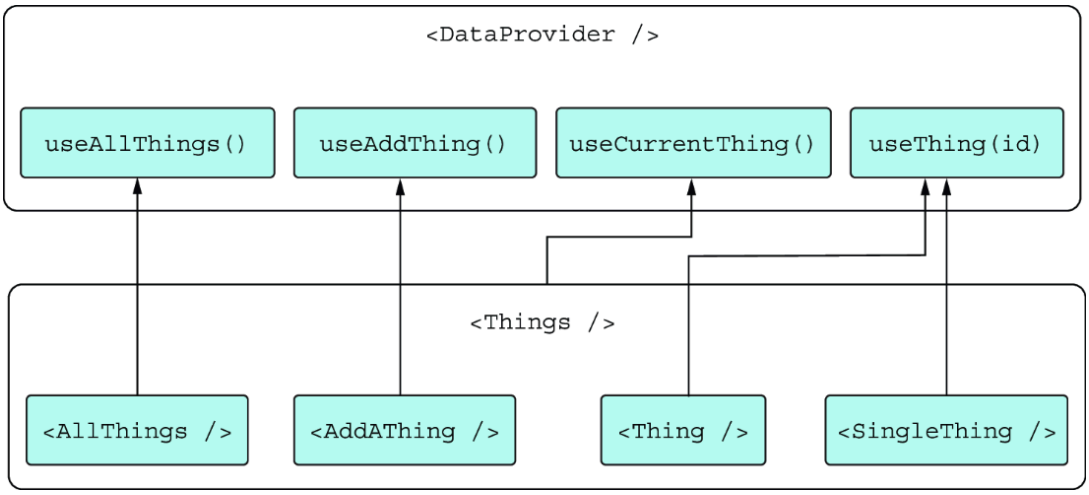

Figure 8.4 The interactions between the presentation layer and the data layer happen in five components, using four custom hooks. The component that displays all things uses the `useAllThings` hook. The component that allows the user to add a thing uses the `useAddThing` hook. Both the component that displays a single thing and the component that displays a thing in a list use the `useThing` hook. Finally, the overall component that displays either a list or a single thing uses the `useCurrentThing` hook.

We expect the following behavior from the four hooks:

- `useCurrentThing` must return either the `id` of the current single thing being displayed or `null` if the list is to be displayed.
- `useAllThings` must return an array of the `id`s of all the things in the order in which they were created.
- `useAddThing` must return a function that accepts a single parameter—the name of a thing to add—and should in turn add that thing to internal data storage.
- `useThing` is the most complex hook. It must return an object with the thing in question (the `id` of which is passed in as a parameter to the hook) and also functions for manipulating said thing: `seeThing`, `seeAllThings`, `doThing`, `undoThing`, `undoLastThing`, and `removeThing`.

The following list describes the latter functions in detail:

- `seeThing()`—Should store the `id` of the thing as the current thing (switching from list view to detail view for that thing)
- `seeAllThings()`—Should clear the `id` of the current thing (switching from detail view to list view).

- `doThing()` —Should append a new entry to the list of the number of times this particular thing has been done
- `undoThing(index)` —Should remove a specific entry (denoted by the index in the list) of the number of times this thing has been done
- `undoLastThing()` —Should remove the last entry in the list of the number of times this thing has been done
- `removeThing()` —Should remove the entire thing and switch to list view if this thing was the current thing

We've built the presentation layer around this simple API. Many of the components are dumb and present only what they've been told to present via props, but a few use this limited API to tap into the data layer to show or manipulate data. Our main obligation from here on out is to create the data layer and make sure that each implementation exports its data and functions according to this API.

One final requirement for the data layer is that we want data to be persisted locally on the computer displaying the application. So if you reload the page, all your data and state are kept intact, including remembering whether you're in list view or detail view. We're adding this requirement because

- It makes the application feel more real because you can reload and your data will still be there.
- It challenges the data layer to come with a built-in solution.

We'll see how the five methods stack up against one another regarding this requirement as well.

8.3 Managing data in pure React

We already know that we can manage data by using `useState` and distribute it throughout our application by using context. In this application, we need two different data items—a list of things and a current thing—so we're going to use the `useState` hook twice. We're also going to need some functions for manipulating the data, which will be wrappers for our state setters.

8.3.1 Context

We're going to shove all this code into a context by using the Provider pattern from chapter 2. Then we can retrieve the relevant data items and data manipulation functions as needed, using the `useContext` hook. Figure 8.5 is a rough outline of how this process is going to work.

What's left is creating all the actions, which are wrappers of `setThings` and `setCurrentThing` as appropriate according to the specifications. When it comes to persisting state to local storage, we have to do that work ourselves. Fortunately, writing data is fairly easy, and `useEffect` is the perfect tool. Whenever the state changes, we persist it to local storage. Reading data is even easier: we initialize the two `useState`s based on local storage data if they exist or reasonable defaults if they don't exist.

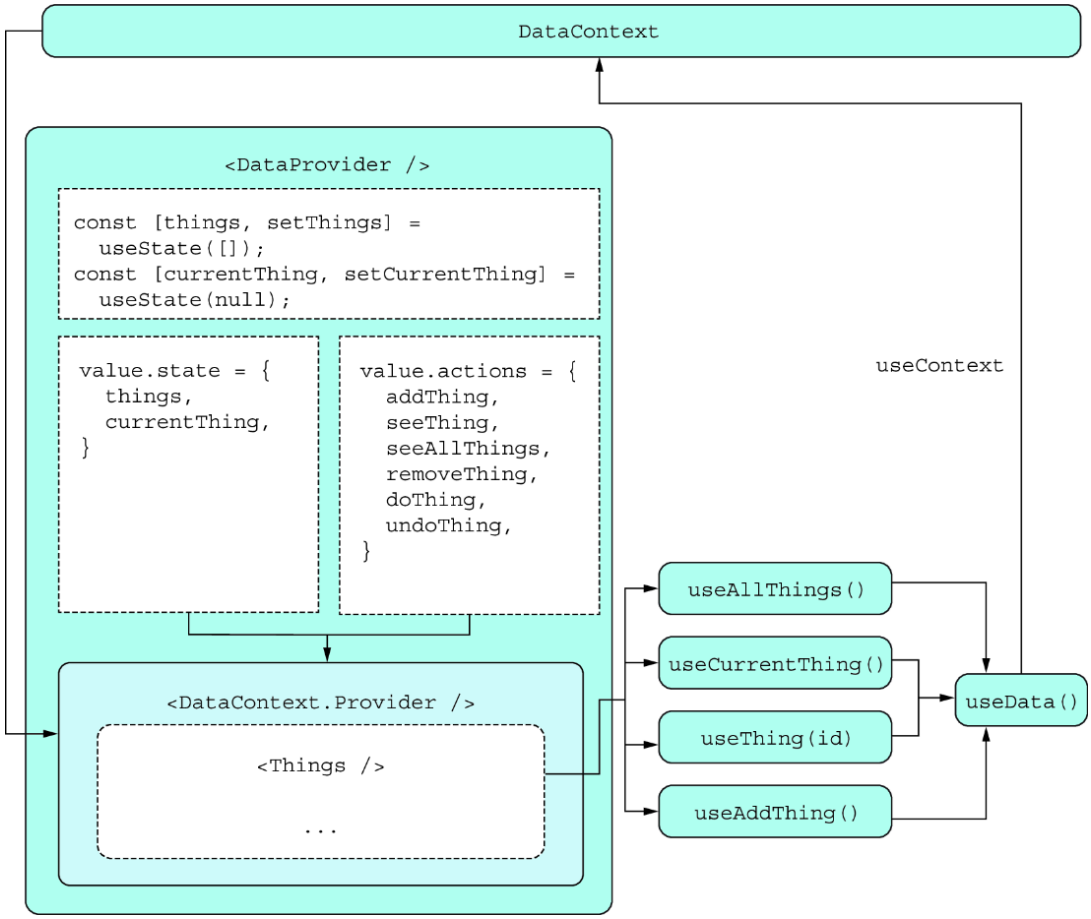

Figure 8.5 We wrap the entire application in a context with the relevant content, and we extract that content by using our custom hooks from said context. The dotted component represents the entire presentation layer, which uses the four hooks. The four hooks in turn use the `useData` hook, consuming the context via `useContext`.

8.3.2 Source code

We have eight files in the data structure, including the index file, but only two files contain anything of significance, and one of them is tiny. All the data storage and manipulation happens in `DataProvider.jsx`, and retrieving data or functions from the context happens in `useData.js`, which is a trivially simple file.

EXAMPLE: CONTEXT

This example is in the `context` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch08/context
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch08-context**.

DATAPROVIDER.JSX

The data provider is the heart of the application, as it both contains and distributes the state to whomever needs it, as well as defines functions that allow components to manipulate the state as needed. See figure 8.6 and listing 8.1.

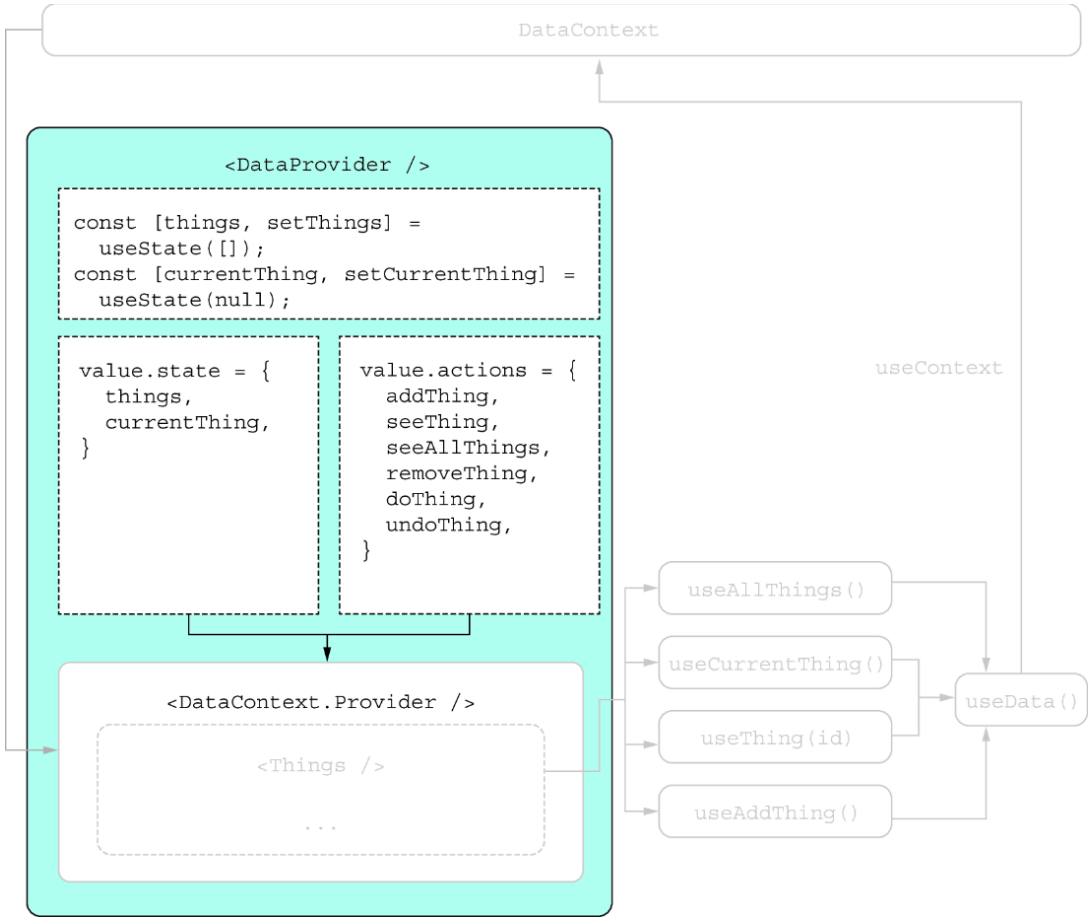

Figure 8.6 The data provider highlighted

Listing 8.1 `src/data/DataProvider.jsx`

```
import { useState, useCallback, useEffect } from "react";
import { v4 as uuid } from "uuid";
import { DataContext } from "./DataContext";
const STORAGE_KEY = "100-things-context";
const INITIAL_STATE =      #1
  JSON.parse(     #1
    localStorage.getItem(STORAGE_KEY)     #1
  ) ||     #1
    { things: [], currentThing: null };     #1
export function DataProvider({ children }) {
  const [things, setThings] =      #2
    useState(INITIAL_STATE.things);     #2
  const [currentThing, setCurrentThing] =    #2
    useState(INITIAL_STATE.currentThing);   #2
    useEffect(     #3
    () =>
      localStorage.setItem(
        STORAGE_KEY,
        JSON.stringify({ things, currentThing })
      ),
    [things, currentThing]
  );
  const addThing = useCallback(      #4
    (name) =>  #4
      setThings((ts) => ts.concat([{ id: uuid(), name, done: [] }])),  #4
    []  #4
  );  #4
  const seeThing = setCurrentThing;     #4
  const seeAllThings = useCallback(     #4
    () => setCurrentThing(null),   #4
    []   #4
  );   #4
  const removeThing = useCallback((id) => {      #4
    setThings((ts) => ts.filter((t) => t.id !== id));    #4
    setCurrentThing((cur) => (cur === id ? null : cur));  #4
  }, []);  #4
  const editThing = useCallback(   #4
    (id, cb) =>   #4
      setThings((ts) =>   #4
        ts.map((t) => (t.id === id ? { ...t, done: cb(t.done) } : t))   #4
      ),   #4
```

```
    []  #4
  );   #4
  const doThing = useCallback(     #4
    (id) => editThing(id, (done) => done.concat(Date.now())),  #4
    [editThing]   #4
  );  #4
  const undoThing = useCallback(     #4
    (id, index) =>
      editThing(id, (done) =>
        done.slice(0, index).concat(done.slice(index + 1))
      ),
    [editThing]
  );
  const value = {     #5
    state: {  #5
      things,   #5
      currentThing, #5
    },  #5
    actions: {  #5
      addThing,  #5
      seeThing,  #5
      seeAllThings,  #5
      doThing,  #5
      undoThing,   #5
      removeThing,  #5
    },  #5
  };  #5
  return (  #5
    <DataContext.Provider value={value}>       #5
      {children}
    </DataContext.Provider>
  );
}
```

#1 First, we load the previous state of the system from local storage if it exists.

#2 Then, we create the two data values, using useState with the defaults.

#3 In an effect in useEffect, we persist any change to either state value to local storage.

#4 The six data manipulation functions are carefully crafted wrappers of the two state setter functions, setThings and setCurrentThing.

#5 Finally, we put all the state and functions inside the context, which we wrap around the rest of the application.

#6 Finally, we put all the state and functions inside the context, which we wrap around the rest of the application.

USEDATA.JS

This hook feeds the other hooks that require values from the context (figure 8.7 and listing 8.2).

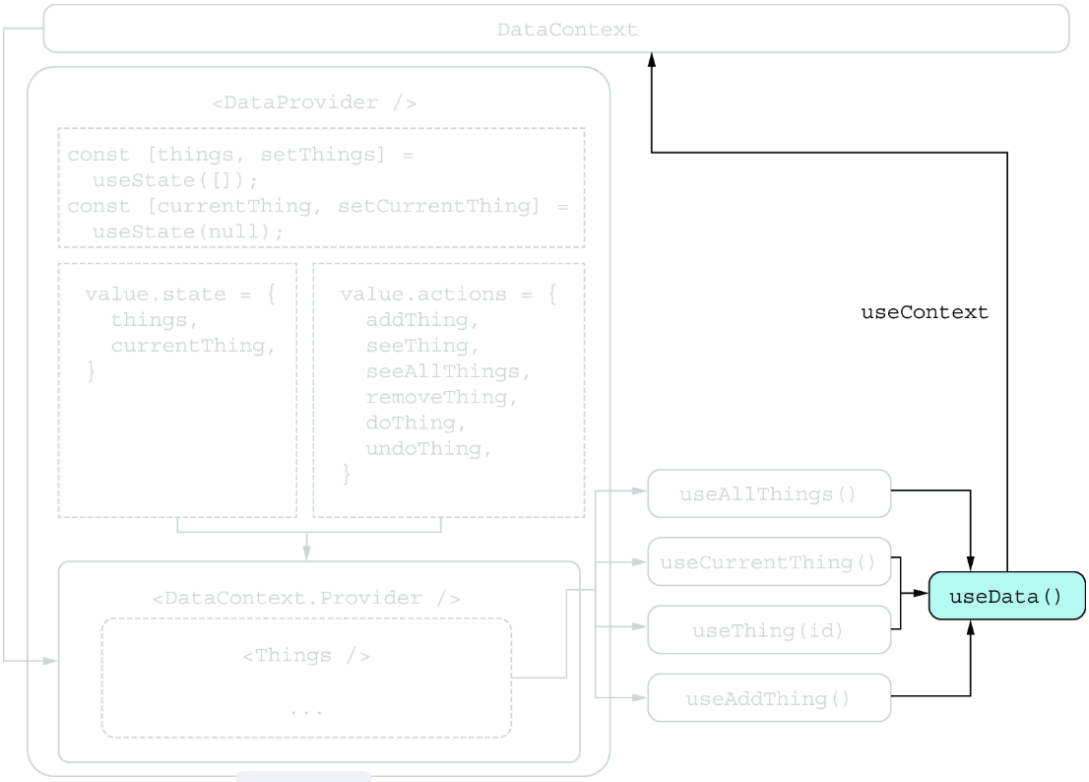

Figure 8.7 The `useData` hook highlighted

Listing 8.2 `src/data/useData.js`

```
import { useContext } from "react";
import { DataContext } from "./DataContext";
export function useData() {
  return useContext(DataContext);    #1
}
```

#1 In this version, we retrieve the entire context, which also means that any component using this hook will re-render any time anything inside the state changes.

DATACONTEXT.JS

This file is the plain context, implemented with React's built-in method (figure 8.8 and listing 8.3).

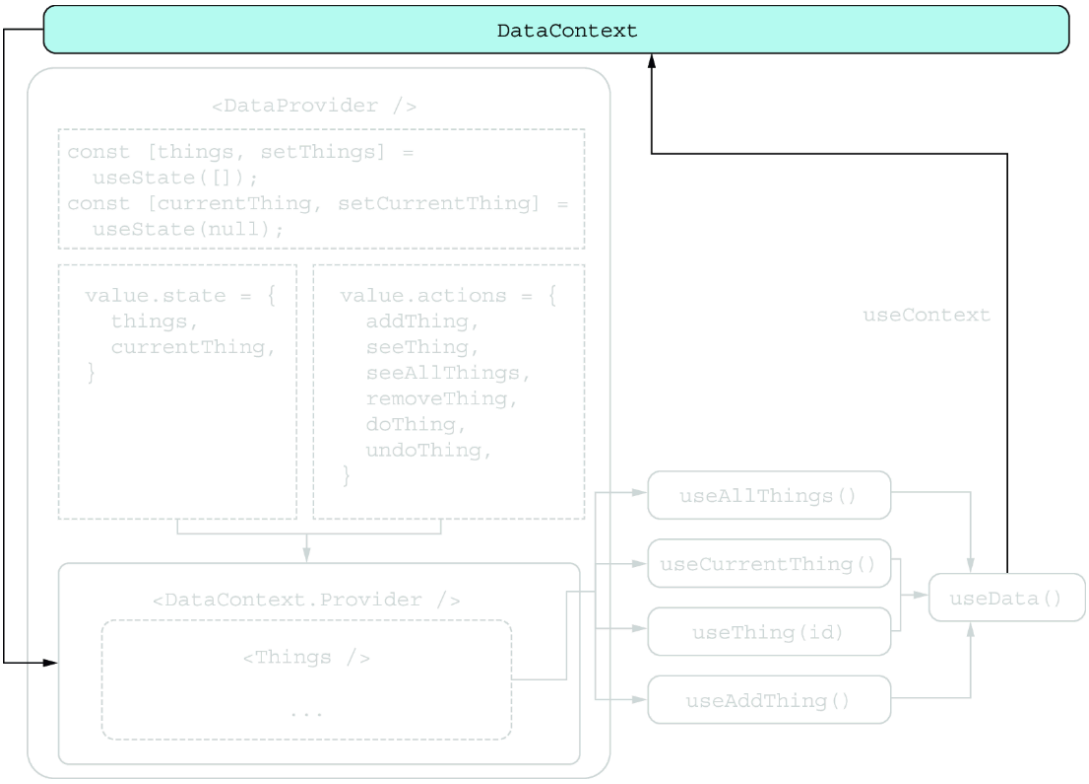

Figure 8.8 The data context highlighted

Listing 8.3 `src/data/DataContext.js`

```
import { createContext } from "react";
export const DataContext =
  createContext({ state: {}, actions: {} });     #1
```

#1 The default context contains no state values or functions.

USEALLTHINGS.JS

When we need to display all the things in our presentation component, we use this hook to extract them from the context (figure 8.9 and listing 8.4).

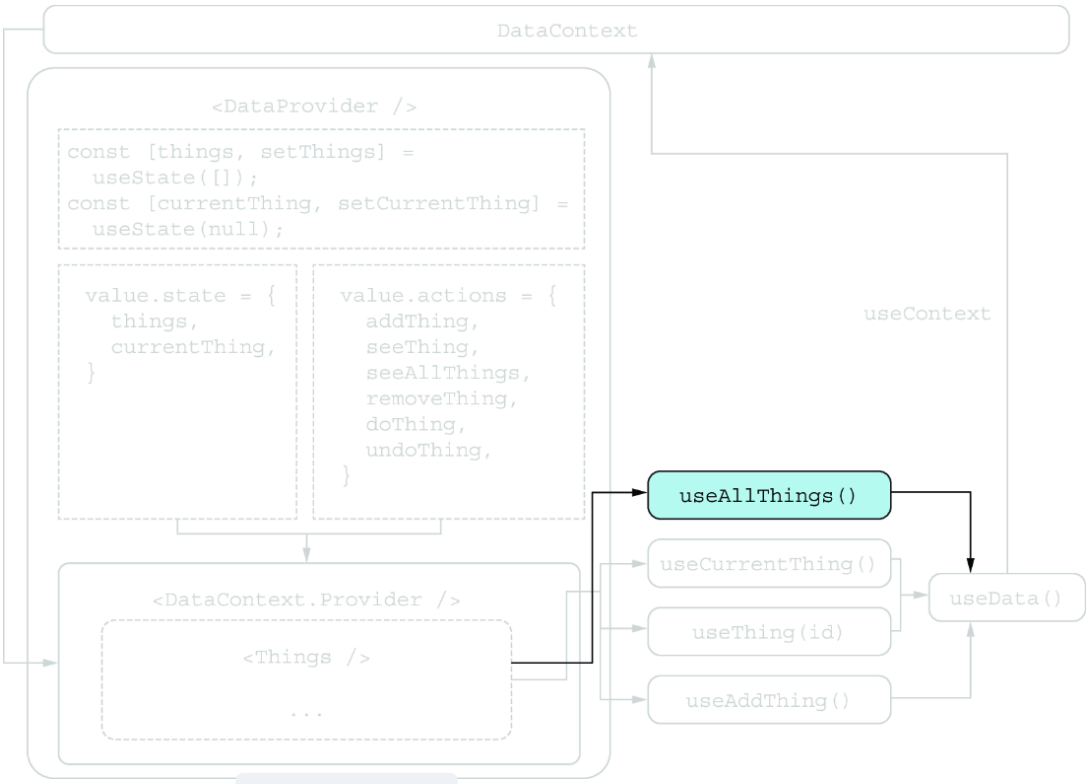

Figure 8.9 The `useAllThings` hook highlighted

Listing 8.4 `src/data/useAllThings.js`

```
import { useData } from "./useData";
export function useAllThings() {
  return useData().state.things     #1
    .map(({ id }) => id);    #1
}
```

#1 Per specification, useAllThings must return an array of ids only, so we map the existing things to get only that property.

USETHING.JS

When we want to interact with a single thing in the presentation layer, we go through the `useThing` hook, passing it the `id` of the thing in question (figure 8.10 and listing 8.5).

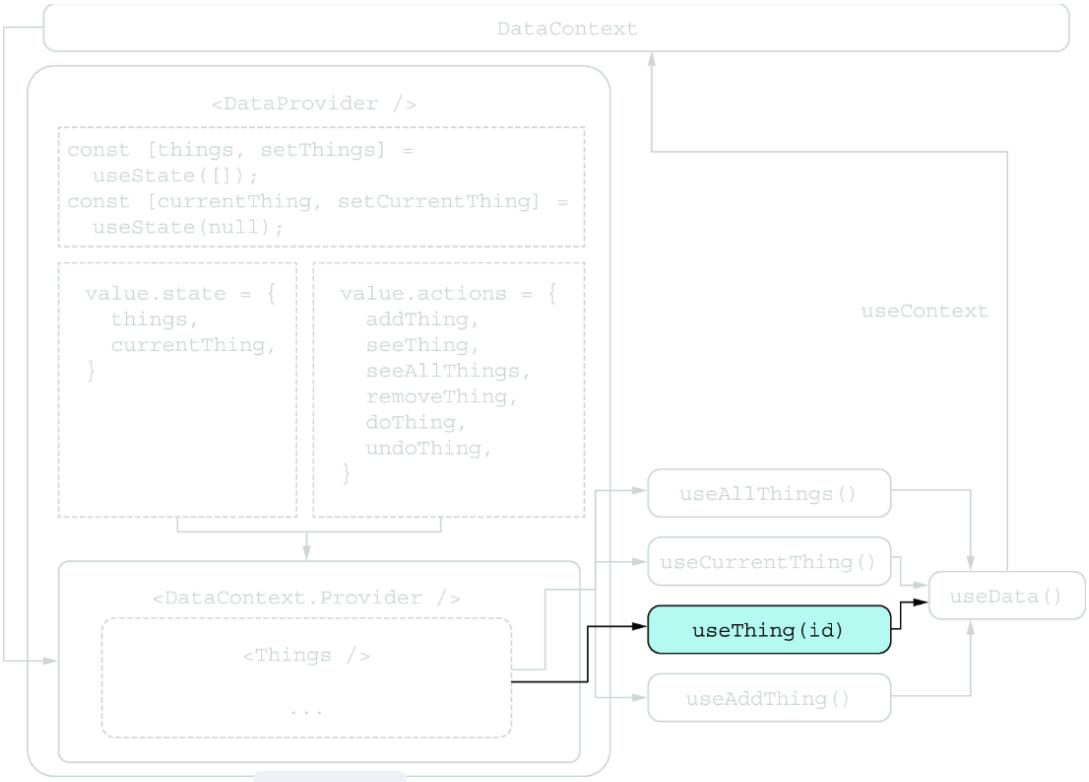

Figure 8.10 The `useThing` hook highlighted

Listing 8.5 `src/data/useThing.js`

```
import { useData } from "./useData";
export function useThing(id) {
  const {
    state: { things },     #1
    actions: {    #1
      seeThing,    #1
      seeAllThings,  #1
      doThing,    #1
      undoThing,   #1
      removeThing,   #1
    },   #1
  } = useData();
  const thing = things.find((t) => t.id === id);    #2
  return {
    thing,
    seeAllThings,
    seeThing: () => seeThing(id),     #3
    removeThing: () => removeThing(id),    #3
    doThing: () => doThing(id),   #3
    undoThing: (index) => undoThing(id, index),     #3
    undoLastThing:    #3
      () => undoThing(id, thing.done.length - 1),    #3
  };
}
```

#1 useThing is the most complex hook, where we have to retrieve the list of things and almost all the functions from the context.
#2 Returns the relevant thing from the entire array
#3 We specialize these five functions with respect to the passed id and return them as specified.
#4 We specialize these five functions with respect to the passed id and return them as specified.

USEADDTHING.JS

Adding a new thing requires invoking the right callback from the context (figure 8.11 and listing 8.6).

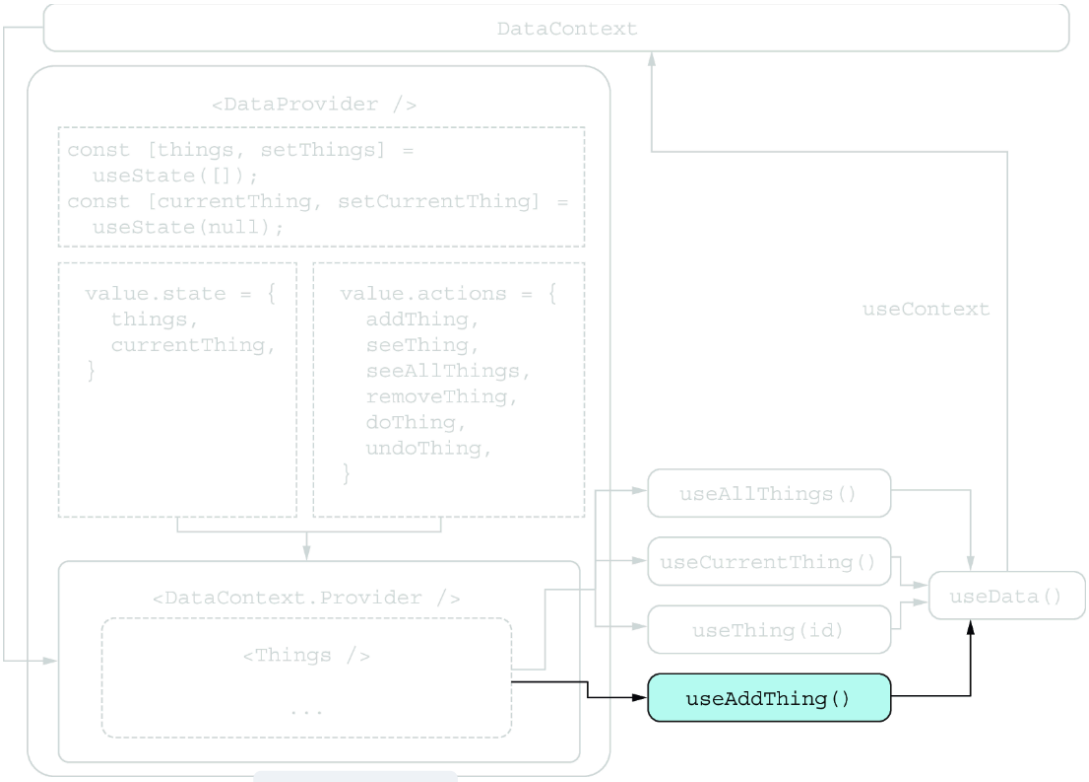

Figure 8.11 The `useAddThing` hook highlighted

Listing 8.6 `src/data/useAddThing.js`

```
import { useData } from "./useData";
export function useAddThing() {
  return useData().actions.addThing;   #1
}
```

#1 This hook returns a single function from the context.

USECURRENTTHING.JS

To determine whether we're displaying list view or detail view, we have this hook to return the `id` of the current thing or `null` if nothing is currently selected (figure 8.12 and listing 8.7).

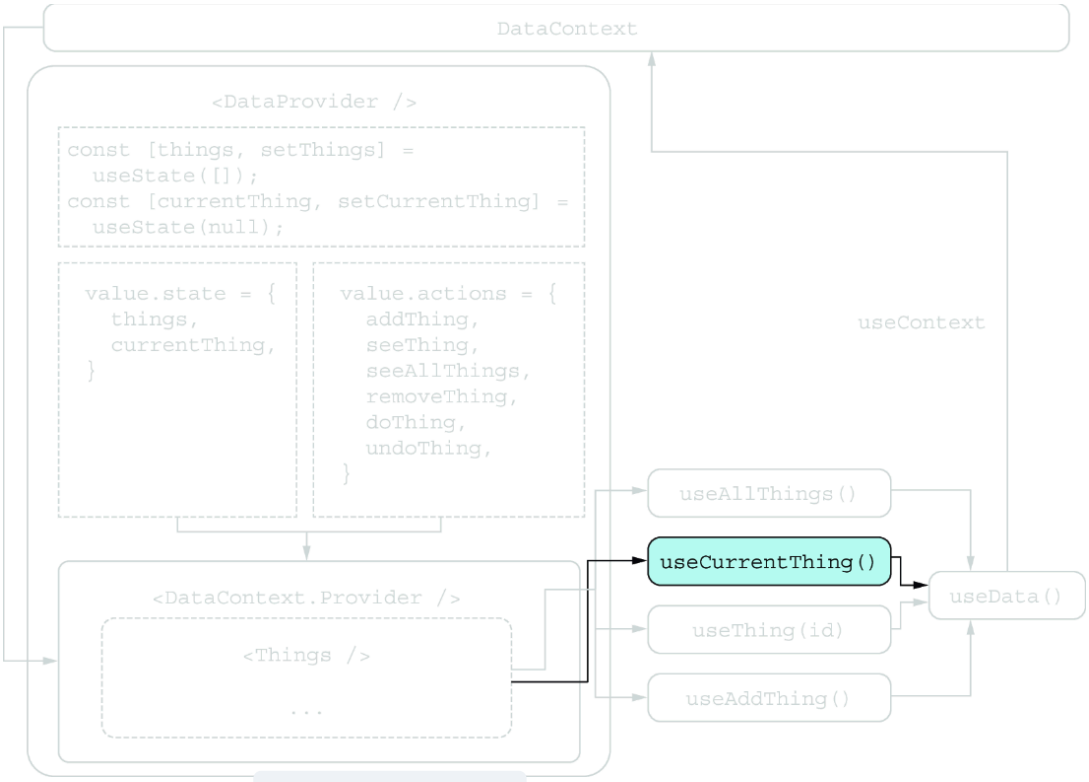

Figure 8.12 The `useCurrentThing` hook highlighted

Listing 8.7 `src/data/useCurrentThing.js`

```
import { useData } from "./useData";
export function useCurrentThing() {
  return useData().state.currentThing;     #1
}
```

#1 Likewise, this hook returns a single state value from the context.

INDEX.JS

This file exports all the externally accessible parts of the data package, as shown in the following listing.

Listing 8.8 `src/data/index.js`

```
export { DataProvider } from "./DataProvider";
export { useAddThing } from "./useAddThing";
export { useAllThings } from "./useAllThings";
export { useThing } from "./useThing";
export { useCurrentThing } from "./useCurrentThing";
```

8.4 Reducing data state

The code in the `DataProvider` in listing 8.1 is a bit messy and spaghetti-like. It can be hard to follow the logic, and it tends to be hard to maintain code like that in the long run. But we've already seen a better way to maintain complex state in chapter 2: the Provider pattern. We're also going to use three external libraries, one of which was also introduced in chapter 2:

- `useReduction`—Allows us to write reducers in React with less boilerplate
- `useContextSelector`—Consumes partial contexts, minimizing re-renders
- *Immer*—Allows us to write immutable code without all the hassle

To get an idea of the overall structure compared with the implementation using `useState` (figure 8.5), see figure 8.13.

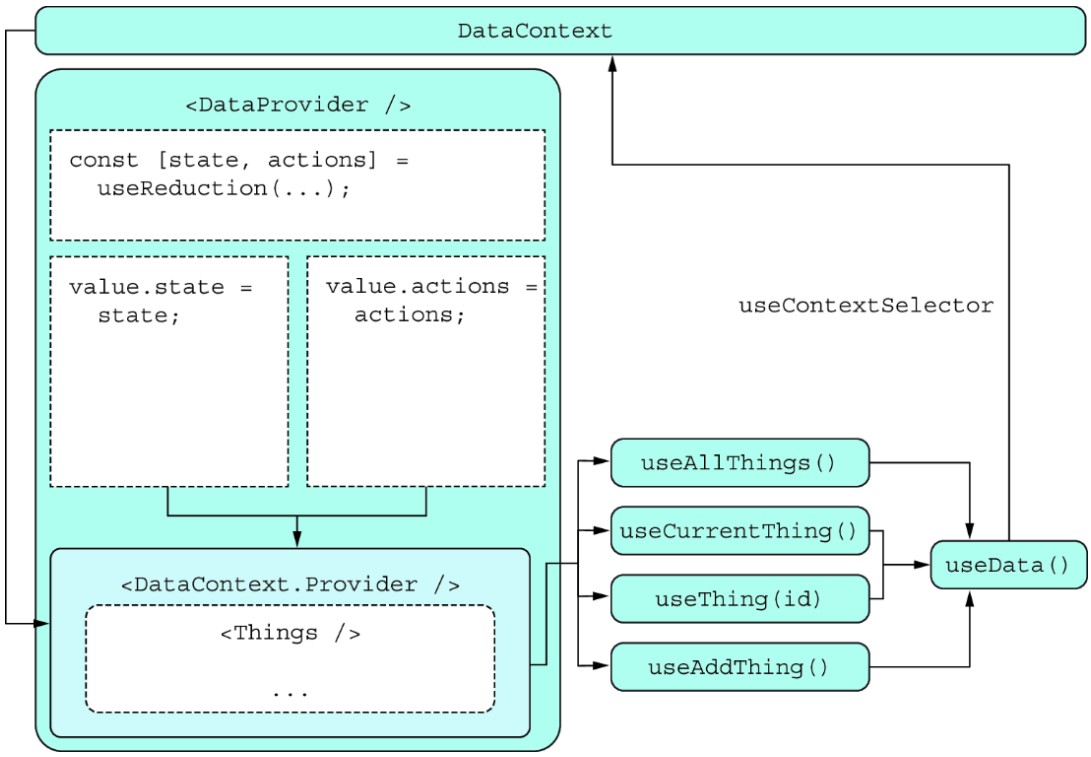

Figure 8.13 The provider and context are used in the same fashion now, but defining the state and the actions is a lot simpler with the `useReduction` package. We also get better performance when we use `useContextSelector` to write the hooks.

8.4.1 Immer: Writing immutable code mutably

In this application, we need to create a reducer that, among other things, handles the `doThing` action. This action takes an `id` of the thing to do and appends a new entry to that thing's list of the number of times this has been done. You can see the structure of the state and what we need to do in figure 8.14.

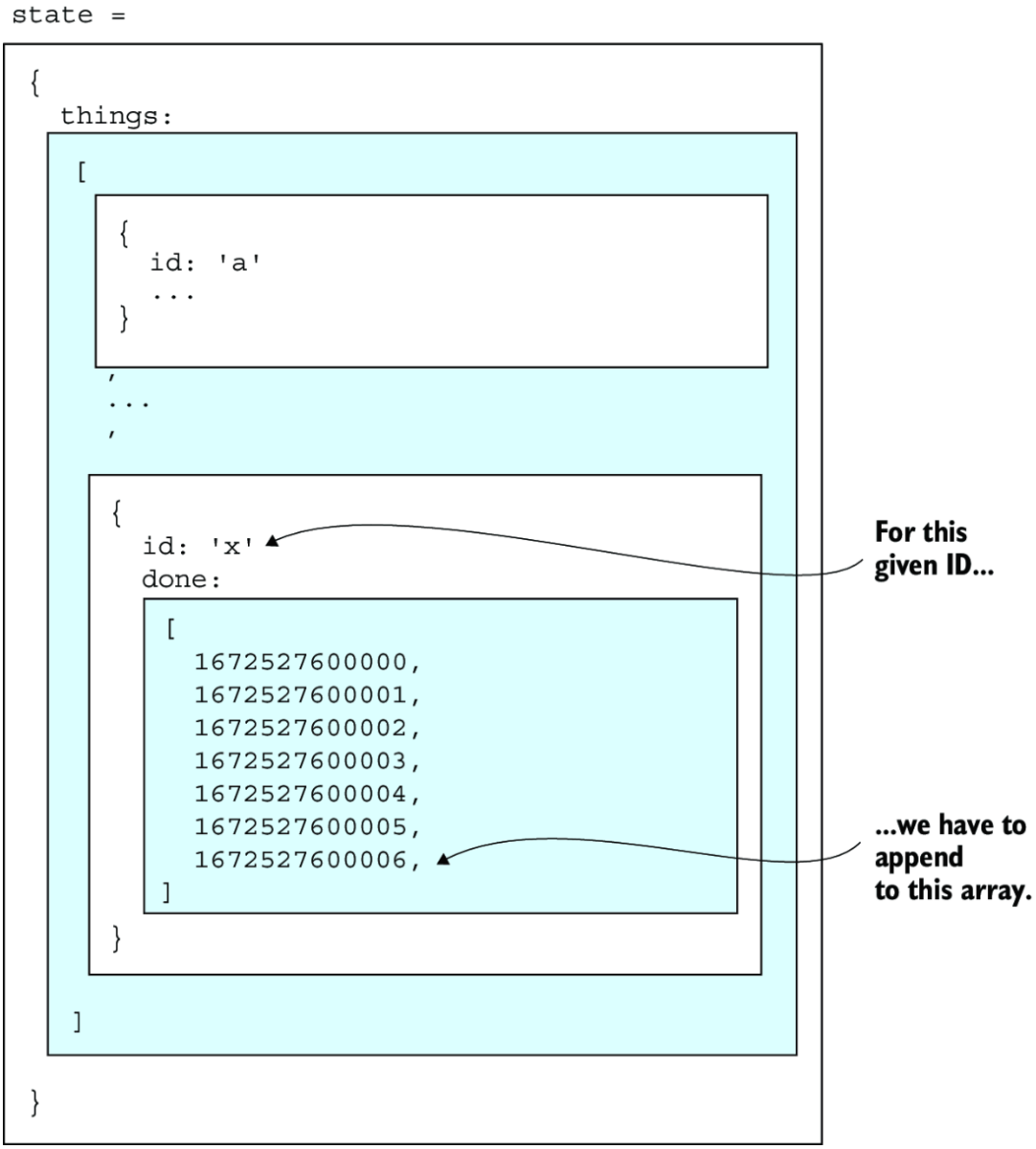

Figure 8.14 We need to modify the innermost array in this nested structure, where the array resides inside an object inside an array inside an object.

If we were to implement the `doThing` function as mutable code, we would do something like this:

```
doThing: (state, {payload: id}) => {
  const thing = state.things.find((thing) => thing.id === id);
  thing.done.push(Date.now());
}
```

Because we're writing an immutable reducer, however, we cannot mutate an array or an object directly, and in the preceding code, we're mutating a nested array directly. Instead, we have to create and use a new array. But because the array exists inside an object, we also have to create a new object, and because that object exists inside another array, we have to create yet another array, which again is inside another object. So this same code in immutable land looks like this:

```
doThing: (state, {payload: id}) => {
  const thingIndex = state.things.findIndex((thing) => thing.id === id);
  const thing = state.things[thingIndex];
  const newThing = {     #1
    ...thing,    #1
    done: thing.done.concat([Date.now()])    #1
  };    #1
  const newThings = [     #2
    ...state.things.slice(0, thingIndex),
    newThing,
    ...state.things.slice(thingIndex + 1),
  ];
  return {...state, things: newThings};    #3
}
```

#1 We create a copy of the object with a copy of the array with an extra item appended.
#2 We put the copy of the object inside a copy of the things array in place of the old one.
#3 Finally, we put the things array inside a copy of the state object.

That's a lot of code compared with the mutable alternative. What if we could get the best of both worlds—write code in the simplest way possible as mutable code but still get the benefits of immutability? That's exactly what Immer does. It allows you to mutate a proxy object directly, and it "plays back" your mutations as immutable alterations of the original object. The process sounds complex, and is (using some sheer

JavaScript magic), but all we need to care about is the fact that it works. With the `produce()` function from Immer, we can write an immutable reducer like so:

```
import produce from 'immer';
...
doThing: produce((draft, {payload: id}) => {   #1
  const thing = draft.things.find((thing) => thing.id === id);
  thing.done.push(Date.now());  #2
})
```

#1 We wrap the function in produce() and accept a draft argument. We use the variable draft to denote that this object is not the actual state object, but a proxy for it that we are allowed to change directly.
#2 We mutate the draft and any objects inside it directly.

I must admit that even I don't fully understand how Immer works on the inside. It uses some kind of proxy object combined with setters and getters to record all the changes being made to the draft object and repeats these changes to the actual state object immutably. To be honest, Immer is kind of like dark matter in the sense that it clearly exists, but we don't know how. Well, I don't, but I guess some very clever people do.

8.4.2 Source code

The structure is almost identical to the version from section 8.3.1, with the same eight files. A lot of those files are different, however, because now we can use the added capabilities we get from using external libraries, making both the application and the code leaner and more efficient. Once again, we need to persist the data manually. But this time, we already have a combined state object, so we serialize that object in and out of local storage without too much hassle.

EXAMPLE: REDUCER

This example is in the `reducer` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch08/reducer
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch08-reducer**.

DATAPROVIDER.JSX

The data provider has the same responsibility as before, but this time, we use a reducer rather than pure `useState` hooks. Otherwise, we expose the same data and callbacks.

Listing 8.9 `src/data/DataProvider.jsx`

```
import { useEffect } from "react";
import { v4 as uuid } from "uuid";
import { produce } from "immer";
import useReduction from "use-reduction";
import { DataContext } from "./DataContext";
const STORAGE_KEY = "100-things-reducer";
const INITIAL_STATE = JSON.parse(localStorage.getItem(STORAGE_KEY)) || {
  things: [],
  currentThing: null,
};
const reducers = {
  seeThing: produce(    #1
    (draft, { payload: newThing }) => {  #1
    draft.currentThing = newThing;  #1
    }  #1
  ),  #1
  seeAllThings: produce((draft) => {     #1
    draft.currentThing = null;  #1
  }), #1
  addThing: produce(    #1
    (draft, { payload: name }) => {  #1
      draft.things.push({ id: uuid(), name, done: [] });  #1
    }  #1
  ),  #1
  removeThing: produce(     #1
    (draft, { payload: id }) => {  #1
      const index = draft.things.findIndex((thing) => thing.id === id);  #1
      if (index !== -1) {  #1
        draft.things.splice(index, 1);  #1
        if (id === draft.currentThing) { #1
          draft.currentThing = null;  #1
        }  #1
      }  #1
    }  #1
  ),  #1
  doThing: produce(     #1
    (draft, { payload: id }) => {  #1
      const thing = draft.things.find((thing) => thing.id === id);  #1
      thing.done.push(Date.now());   #1
    }  #1
  ),  #1
```

```
    undoThing: produce(      #1
      (draft, { payload: { id, index } }) => {  #1
        const thing = draft.things.find((thing) => thing.id === id);
        thing.done.splice(index, 1);
      }
    ),
  };
  export function DataProvider({ children }) {
    const [state, actions] = useReduction(INITIAL_STATE, reducers);
    useEffect(
      () => localStorage.setItem(STORAGE_KEY, JSON.stringify(state)),
      [state]
    );
    const value = { state, actions };
    return (
      <DataContext.Provider value={value}>
        {children}
      </DataContext.Provider>
    );
  }
```

#1 This time, we're creating all the data manipulation functions as reducers, using Immer's produce() function.
#2 This time, we're creating all the data manipulation functions as reducers, using Immer's produce() function.

USEDATA.JS

We still have this intermediate hook to access values in the context, but this time, we use the `useContext` hook from the `useContextSelector` library.

Listing 8.10 `src/data/useData.js`

```
import { useContextSelector } from "use-context-selector";
import { DataContext } from "./DataContext";
export function useData(selector) {
  return useContextSelector(     #1
    DataContext,    #1
    selector,   #1
  );    #1
}
```

#1 Accessing the context via a selector for improved performance

DATACONTEXT.JS

Similarly, we have to define the context by using the `useContextSelector` library, as shown in the following listing.

Listing 8.11 `src/data/DataContext.js`

```
import {     #1
  createContext,    #1
} from "use-context-selector";    #1
export const DataContext = createContext({});
```

#1 Here, we remember to create the context by using the createContext function from the third-party library, not the default one in React itself.

USEALLTHINGS.JS

Accessing values from the context is much cleaner when we use a selector.

Listing 8.12 `src/data/useAllThings.js`

```
import { useData } from "./useData";
export function useAllThings() {
  return useData( ({ state }) => state.things)     #1
    .map(({ id }) => id);    #1
}
```

#1 We select the array as is inside the selector and manipulate it outside for optimization reasons. If the array doesn't change, doing it this way doesn't cause a re-render. If we did the mapping inside the selector, we would always generate a new array, so we would always re-render.

USETHING.JS

When we need to access multiple values from the context, we can do so in separate calls to the `useData` hook.

Listing 8.13 `src/data/useThing.js`

```
import { useData } from "./useData";
export function useThing(id) {
  const thing = useData(({ state }) =>     #1
    state.things.find((t) => t.id === id)     #1
  );    #1
  const {   #1
    seeThing,    #1
    seeAllThings,   #1
    doThing,   #1
    undoThing,    #1
    removeThing,    #1
  } = useData(({ actions }) => actions);   #1
  return {
    thing,
    seeAllThings,
    seeThing: () => seeThing(id),
    removeThing: () => removeThing(id),
    doThing: () => doThing(id),
    undoThing: (index) => undoThing({ id, index }),
    undoLastThing: () =>
      undoThing({ id, index: thing.done.length - 1 }),
  };
}
```

#1 Because we can select only one thing at a time by using the default selector function, we split the selector into two parts. Remember that the whole actions object can be considered stable, as it is returned in full by the reducer hook.

USEADDTHING.JS

This file is another simple value retrieval from the context.

Listing 8.14 `src/data/useAddThing.js`

```
import { useData } from "./useData";
export function useAddThing() {
  return useData(({ actions }) => actions.addThing);
}
```

USECURRENTTHING.JS

This file is yet another simple value retrieval from the context.

Listing 8.15 `src/data/useCurrentThing.js`

```
import { useData } from "./useData";
export function useCurrentThing() {
  return useData(({ state }) => state.currentThing);
}
```

INDEX.JS

This file is identical to the first iteration.

Listing 8.16 `src/data/index.js`

```
export { DataProvider } from "./DataProvider";
export { useAddThing } from "./useAddThing";
export { useAllThings } from "./useAllThings";
export { useThing } from "./useThing";
export { useCurrentThing } from "./useCurrentThing";
```

8.5 Scaling data management with Redux Toolkit

Because we know how the React hook `useReducer` works, we already know how Redux works. Redux is the source of the `useReducer` functionality and came long before hooks were ever conceived of.
Originally, Redux was rather boilerplate heavy, and creating a simple application with a Redux store often required creating a ton of configuration and functions that were almost trivial. For all the trivial cases, Redux comes in a much more compact edition known as RTK, which works almost exactly like `useReduction`, which we've just seen in section 8.4. `useReduction` is also based on how RTK works. So although we haven't used Redux in this book, we've used tools based on it, so we can get started easily.

8.5.1 How does Redux work?

To understand Redux, it's important to understand the Redux cycle. Redux has a store with reducers that are invoked from action creators dispatching actions. You can see the cycle depicted in figure 8.15.

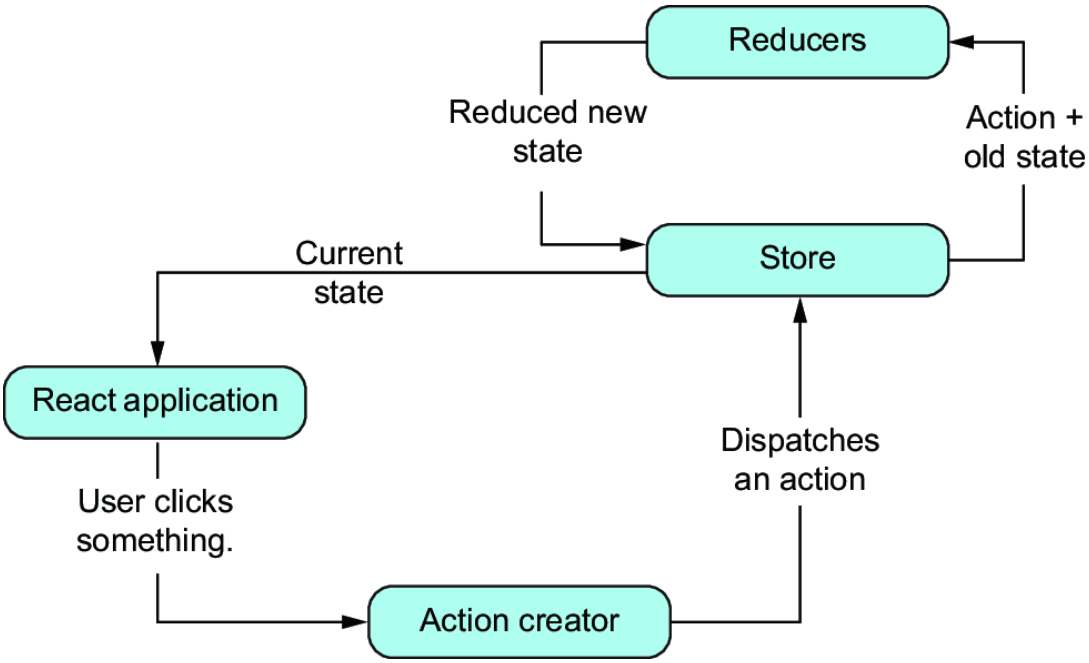

Figure 8.15 The simplified Redux cycle. State is kept in the store. The state defines how the application is rendered. When the user interacts with the application, actions are dispatched, and the store sends them to the relevant reducer along with the old state. The newly reduced state is updated in the store and sent back to the application, and the cycle repeats.

In classic Redux, you define action creators, actions, and reducers separately; in RTK, you define them all at the same time. So if you define a reducer named `increment`, you automatically get an action creator named `increment()`, which dispatches an action that invokes that reducer.

The Redux store is split into slices. Each slice consists of its own reducers, actions, and action creators. What makes up a slice is a decision made by the developer. Normally, we put related items in the same slice. If we added a menu that could be opened and a dark mode toggle, we could put those values in a `ui` slice. If we added authentication, we could put user data and login status in an `auth` slice. For this application, we have only a single slice of data, so we will call that slice `data`.

When we apply the Redux methodology to our data layer, we get what you see in figure 8.16. Redux comes with its own provider (which internally uses a React context), so we do not need to create our own context. Redux also comes with its own selector hook, `useSelector`, so we don't need to use the `useContextSelector` module.

NOTE To be honest, Redux is basically the same as context + `useReduction` + `useContextSelector`.

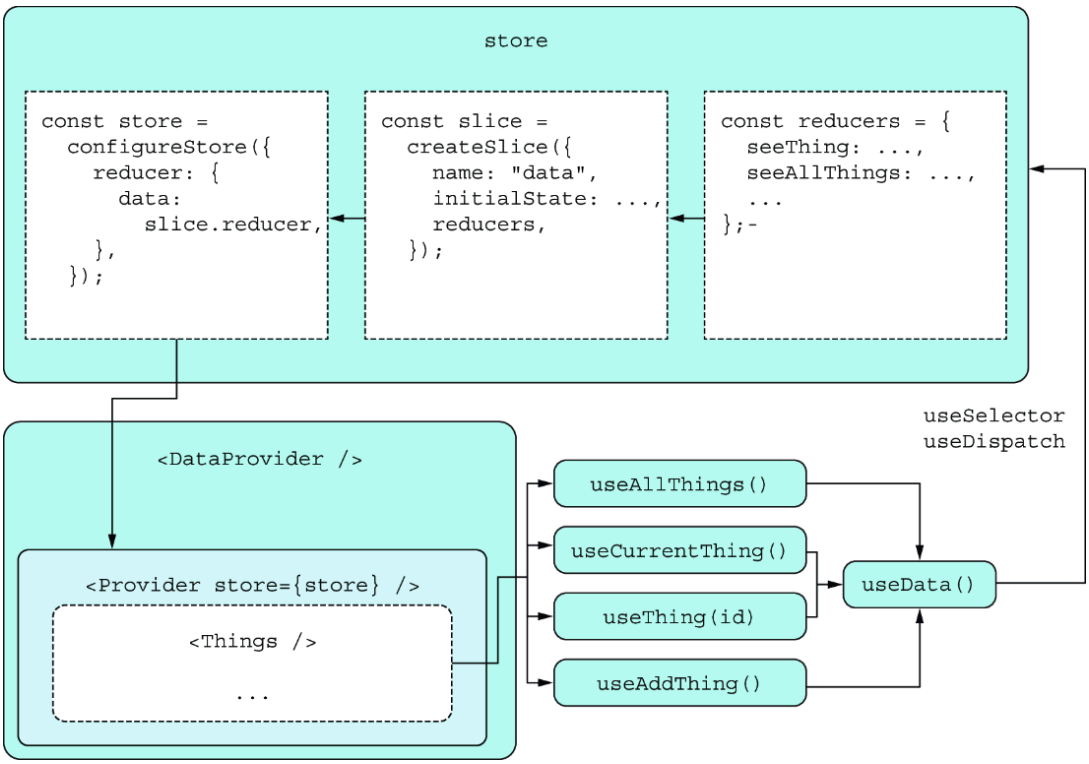

Figure 8.16 The data layer looks like this when we use Redux. Redux comes with a built-in provider, and `useSelector` and `useDispatch` hooks, which we'll use in our four main hooks to read and update state, respectively.

8.5.2 Source code

RTK has Immer built into its reducers, so we can more or less copy and paste the reducers from the previous implementation, as they are essentially identical and work the same way. To read the persisted state, we need to read the state from local storage manually as we define the slice of data. To write state, we need to listen for updates to the store, using the aptly named method `store.subscribe`, which takes a callback that triggers every time anything inside the store updates. Listing 8.18 shows how we can use this function to store the state.

EXAMPLE: REDUX

This example is in the `redux` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch08/redux
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch08-redux**.

STORE.JS

The store is the central hub for this application—where we define the main reducer, which manipulates the state as required. Note that this process is similar to using `useReduction` to create a reducer.

Listing 8.17 `src/data/store.js`

```
import { v4 as uuid } from "uuid";
import { createSlice, configureStore } from "@reduxjs/toolkit";
export const STORAGE_KEY = "100-things-redux";
const getInitialThings = () =>
  JSON.parse(localStorage.getItem(STORAGE_KEY)) || {
    things: [],
    currentThing: null,
  };
const reducers = {
  seeThing: (draft, { payload: newThing }) => {    #1
    draft.currentThing = newThing;   #1
  },   #1
  seeAllThings: (draft) => {    #1
    draft.currentThing = null;    #1
  },   #1
  addThing: (draft, { payload: name }) => {    #1
    draft.things.push({ id: uuid(), name, done: [] });   #1
  },   #1
  removeThing: (draft, { payload: id }) => {     #1
    const index = draft.things.findIndex((thing) => thing.id === id);   #1
    if (index !== -1) {   #1
      draft.things.splice(index, 1);   #1
      if (id === draft.currentThing) {    #1
        draft.currentThing = null;    #1
      }   #1
    }   #1
  },   #1
  doThing: (draft, { payload: id }) => {     #1
    const thing = draft.things.find((thing) => thing.id === id);   #1
    thing.done.push(Date.now());   #1
  },   #1
  undoThing:   #1
   (draft, { payload: { id, index } }) => {    #1
    const thing = draft.things.find((thing) => thing.id === id);
    thing.done.splice(index, 1);
  },
};
const dataSlice = createSlice({     #2
  name: "data",
  initialState: getInitialThings(),
  reducers,
```

```
});

export const store = configureStore({    #3
  reducer: {
    data: dataSlice.reducer,
  },
});
export const actions = dataSlice.actions;     #4
```

#1 We create the reducers' mutating state without a worry because we know that they will be wrapped with Immer.
#2 The reducers are passed into our main slice, which also has a name and an initial value loaded from local storage.
#3 That slice is passed to our store, which is exported for the data provider.
#4 Finally, we also export the actions as defined in our slice.

DATAPROVIDER.JSX

The data provider doesn't have to do anything in this iteration except provide the store from Redux. Because we also need to persist the state to local storage, however, we do that here.

Listing 8.18 `src/data/DataProvider.jsx`

```
import { useEffect } from "react";
import { store, STORAGE_KEY } from "./store";
import { Provider } from "react-redux";     #1
export function DataProvider({ children }) {
  useEffect(
    () =>
      store.subscribe(() =>     #2
        localStorage.setItem(
          STORAGE_KEY,
          JSON.stringify(store.getState().data)
        )
      ),
    []
  );
  return (    #3
    <Provider store={store}> #3
      {children}   #3
    </Provider>  #3
  );   #3
}
```

#1 We use the provider given to us by the react-redux package, which is a wrapper for a regular React context provider.
#2 To persist data on change to local storage, we subscribe to the store. Note that this subscribe method returns an unsubscribe function, which the effect also returns, so it will clean up after itself.
#3 We put the store and the provider together surrounding the entire application.

USEDATA.JS

To read data from the store, we use the Redux selector hook.

Listing 8.19 `src/data/useData.js`

```
import { useSelector } from "react-redux";
export function useData(selector) {
  return useSelector(selector);     #1
}
```

#1 Redux comes with its own selector function, so we use that function.

USEALLTHINGS.JS

This file retrieves a single value from the store.

Listing 8.20 `src/data/useAllThings.js`

```
import { useData } from "./useData";
export function useAllThings() {
  return useData((store) => store.data.things)    #1
    .map(({ id }) => id);
}
```

#1 The important thing to remember here is that we read our values from the data slice of the store; thus, we need to access store.data.

USETHING.JS

Here, we need to access data from the store and dispatch actions to the store.

Listing 8.21 `src/data/useThing.js`

```
import { useData } from "./useData";
import { actions } from "./store";
import { useDispatch } from "react-redux";
export function useThing(id) {
  const thing = useData((store) =>
    store.data.things.find((t) => t.id === id)
  );
  const dispatch = useDispatch();     #1
  return {
    thing,
    seeThing: () => dispatch(actions.seeThing(id)),
    removeThing: () => dispatch(actions.removeThing(id)),
    doThing: () => dispatch(actions.doThing(id)),
    seeAllThings: () => dispatch(actions.seeAllThings()),
    undoThing: (index) => dispatch(actions.undoThing({ id, index })),
    undoLastThing: () =>
      dispatch(actions.undoThing({ id, index: thing.done.length - 1 })),
  };
}
```

#1 The only thing new here is that to dispatch an action, we need the dispatch function from Redux, which needs access to the provider, so it has to be retrieved from a hook.

USEADDTHING.JS

We need the dispatch method to create the correct action.

Listing 8.22 `src/data/useAddThing.js`

```
import { actions } from "./store";
import { useDispatch } from "react-redux";
export function useAddThing() {
  const dispatch = useDispatch();
  return (name) => dispatch(actions.addThing(name));
}
```

USECURRENTTHING.JS

This file retrieves a single value from the store.

Listing 8.23 `src/data/useCurrentThing.js`

```
import { useData } from "./useData";
export function useCurrentThing() {
  return useData((store) => store.data.currentThing);
}
```

INDEX.JS

This file is identical to the last version.

Listing 8.24 `src/data/index.js`

```
export { DataProvider } from "./DataProvider";
export { useAddThing } from "./useAddThing";
export { useAllThings } from "./useAllThings";
export { useThing } from "./useThing";
export { useCurrentThing } from "./useCurrentThing";
```

8.6 Simplifying data management with zustand

To make the reducer-based variant in section 8.4 work, we needed five things:

- The reducer functions
- A hook to generate new state when actions are dispatched to the reducer (`useReducer` or `useReduction`)
- A context to hold the state and actions
- A context provider to distribute the data
- A context consumer hook to read the state and retrieve the actions from (`useContext` or `useContextSelector`)

Suppose that we take away the last three items in the preceding list, leaving only the reducer functions and the main reducer hook. That list would be zustand in a nutshell.

8.6.1 Zustand

Zustand comes with a single function, `create()`, which returns a hook that we can use wherever we want. No matter where we use the hook, it works on the same state and contains the same actions reducing said state. If we take away all the context bits and have only a single hook, `useData`, that we create with the zustand API, what we have left is the super-simplified data layer in figure 8.17.

Zustand even comes bundled with Immer, but you don't have to use it. Using Immer in zustand, we apply it as middleware, which means that we don't need to curry each function with `produce()`. We say that the entire reducer object is using Immer-style reducers, and they will be wrapped correctly.

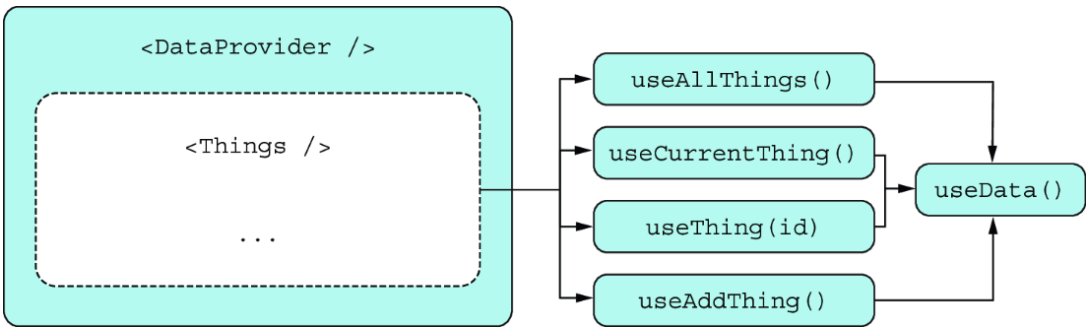

Figure 8.17 With zustand, the data layer becomes extra simple. We have only a single hook that all our hooks go through. There's no context to provide or consume; the state lives inside the hook no matter where we use it. I've displayed the data provider with a dashed outline because it doesn't do anything anymore. I keep it there only because it makes this variant comparable to the others.

8.6.2 Source code

We have only seven files this time, but we still have four public hooks and an index file. The provider does nothing (but we include it so that the data layer API is identical to the other examples), and everything happens in the `useData` hook.

Finally, we need to tackle how to persist the data to local storage. But zustand has even that capability built in, so we can use the `persist()` function from the zustand middleware library and pass our reducer into that function with the relevant storage key.

EXAMPLE: ZUSTAND

This example is in the `zustand` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch08/zustand
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch08-zustand**.

USEDATA.JS

This file is where all the magic happens in this implementation. In this file, we create a hook by calling zustand, which returns the hook we need. We pass in both the state values and the callbacks we need. All the callbacks use Immer for easier state manipulation.

Listing 8.25 `src/data/useData.js`

```
import { create } from "zustand";
import { v4 as uuid } from "uuid";
import { persist } from "zustand/middleware";
import { immer } from "zustand/middleware/immer";
export const useData = create(    #1
  persist(     #2
    immer((set) => ({   #2
      things: [],
      currentThing: null,
      seeThing: (newThing) =>
        set((draft) => {
          draft.currentThing = newThing;
        }),
      seeAllThings: () =>
        set((draft) => {
          draft.currentThing = null;
        }),
      addThing: (name) =>
        set((draft) => {
          draft.things.push({ id: uuid(), name, done: [] });
        }),
      removeThing: (id) =>
        set((draft) => {
          const index = draft.things.findIndex((thing) => thing.id === id);
          if (index !== -1) {
            draft.things.splice(index, 1);
            if (id === draft.currentThing) {
              draft.currentThing = null;
            }
          }
        }),
      doThing: (id) =>
        set((draft) => {
          const thing = draft.things.find((thing) => thing.id === id);
          thing.done.push(Date.now());
        }),
      undoThing: (id, index) =>
        set((draft) => {
          const thing = draft.things.find((thing) => thing.id === id);
          thing.done.splice(index, 1);
        }),
```

```
    })),
    { name: "100-things-zustand" }
  )
);
```

#1 We create a hook by calling a function that returns a hook. We haven't seen this type of structure before, and it is fairly advanced, but hook generators are apparently a thing.
#2 We use two pieces of middleware, one to persist the data to local storage and another to curry the reducers with Immer. We could have applied this middleware in any order.

DATAPROVIDER.JSX

The data provider doesn't do anything in this version; it's here only so that this iteration has the same interface as the other ones. The only interesting bit is that we import the `useData` hook without using it. Importing the hook makes React initialize the hook with the initial data, as this file will be the first one to include this import.

Listing 8.26 `src/data/DataProvider.jsx`

```
import "./useData";    #1
export function DataProvider({ children }) {
  return children;     #2
}
```

#1 We import the data hook to make sure that it has been initialized—not strictly necessary, only a precaution.
#2 The provider does nothing but return the children.

USEALLTHINGS.JS

Accessing data from the stateful hook is easy.

Listing 8.27 `src/data/useAllThings.js`

```
import useData from "./useData";
export function useAllThings() {
  return useData((state) => state.things).map(({ id }) => id);
}
```

USETHING.JS

This hook accesses data and callbacks from the context.

Listing 8.28 `src/data/useThing.js`

```
import { useData } from "./useData";
export function useThing(id) {
  const thing =     #1
    useData(    #1
      (state) =>    #1
        state.things.find((t) => t.id === id)   #1
    );   #1
  const seeThing =    #1
    useData((state) => state.seeThing);    #1
  const seeAllThings =   #1
    useData((state) => state.seeAllThings);    #1
  const doThing =   #1
    useData((state) => state.doThing);    #1
  const undoThing =   #1
    useData((state) => state.undoThing);   #1
  const removeThing =    #1
    useData((state) => state.removeThing);   #1
  return {
    thing,
    seeThing: () => seeThing(id),
    removeThing: () => removeThing(id),
    doThing: () => doThing(id),
    seeAllThings,
    undoThing: (index) => undoThing(id, index),
    undoLastThing: () => undoThing(id, thing.done.length - 1),
  };
}
```

#1 By default, zustand lets you select only one thing from the store, but we need six things. We can modify our store to allow the selection of multiple bits, but the effort isn't worthwhile, as we'd need this feature only once. So we make six separate selections, and we're good.

USEADDTHING.JS

This file accesses a single callback from the hook.

Listing 8.29 `src/data/useAddThing.js`

```
import { useData } from "./useData";
export function useAddThing() {
  return useData((state) => state.addThing);
}
```

USECURRENTTHING.JS

This file retrieves data from the hook.

Listing 8.30 `src/data/useCurrentThing.js`

```
import { useData } from "./useData";
export function useCurrentThing() {
  return useData((state) => state.currentThing);
}
```

INDEX.JS

This file is identical to the last iteration.

Listing 8.31 `src/data/index.js`

```
export { DataProvider } from "./DataProvider";
export { useAddThing } from "./useAddThing";
export { useAllThings } from "./useAllThings";
export { useThing } from "./useThing";
export { useCurrentThing } from "./useCurrentThing";
```

8.7 Rethinking flow and data with XState

State machines enable us to define the flow of a (complex) operation. Suppose that you are creating the signup section for a website. For this particular website, the user has to input first their email and password. On the next screen, they can enter more information, such as name and birth date, but they can skip this screen. At the end, they have to accept the website terms. Also, they can go backward through the flow. The

state can flow in many ways, and creating a state diagram with associated actions can help us visualize what can happen when (figure 8.18).

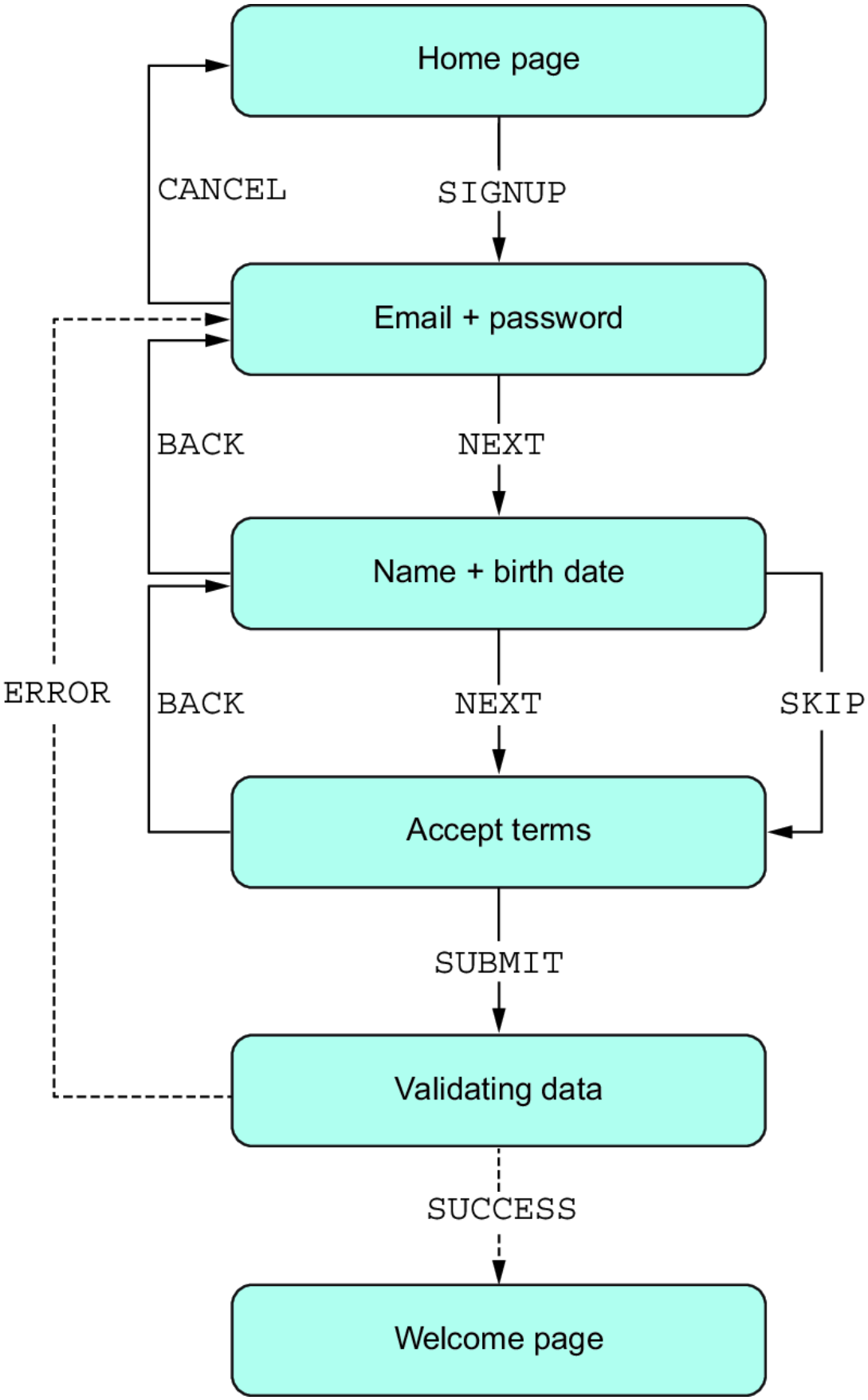

Figure 8.18 As the visitor must have a user to proceed on a given website, they can navigate the different states of the diagram, and in the end, they will be given access to the full website. States are boxes, and transitions are arrows. Only the arrows leading out from a given state are allowed transitions from that state. Fully drawn arrows are user-initiated transitions, and dashed arrows are automatic transitions based on various criteria.

This arrangement seems great for state and flow management, but how can we use it for data management? Well, we can have contextual data that exists outside any particular state, belongs to the overall flow, and is manipulated as transitions occur. Maybe a user logging in can type their password wrong only three times, after which no more retries are allowed. That data would be contextual data that lives above the state diagram, not inside any single state or transition. We can use this feature of a state machine to our advantage so we can keep our business data in the machine context.

8.7.1 A state machine for doing things

Let's see how we would use a state machine for our application. First, we want to create a state diagram for this application. We have two different states: list view and detail view. We also have different transitions, but most of them don't change the state (in state-machine terms). When we add a new thing, for example, we manipulate data, but we don't change the state; we stay in list view. The only times we change the state are when we want to see a thing (change from list view to detail view) and when we want to see all the things or remove a thing (move from detail view to list view). You can see the first iteration in figure 8.19.

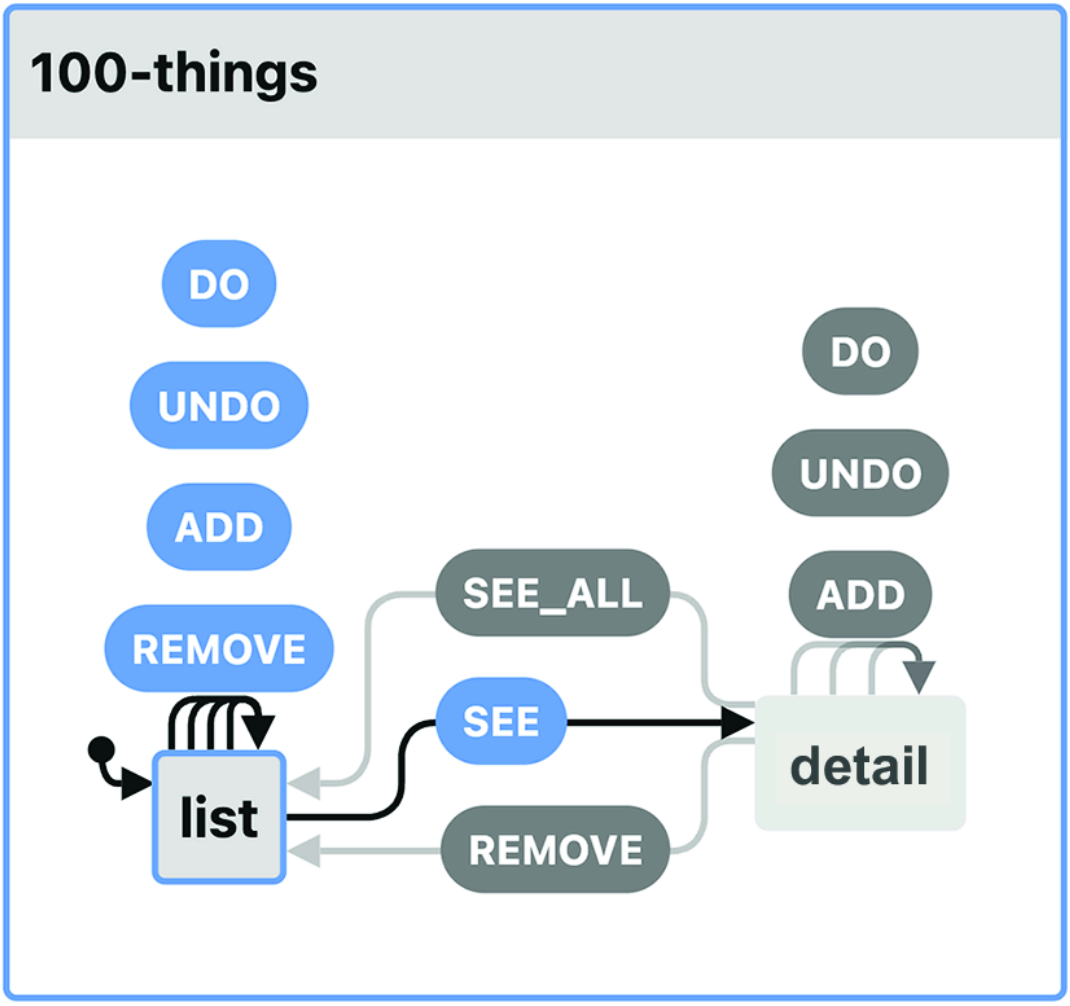

Figure 8.19 In the first iteration of our state flowchart, we are in list view, and we can do five different transitions, only one of which leads to detail view.

Looking at this state diagram more closely, we notice some transitions that don't exist. We can add things only in list view, not in detail view. We don't have a button for adding a new thing in detail view, so the dark-gray `ADD` transition on the right shouldn't exist in detail view. Similarly, we cannot remove a thing from list view—only from detail view. Figure 8.20 cleans up these extra transitions.

You may wonder where this visualization comes from because it's not like others in this book. The source is an online tool called Stately (**https://stately.ai**), which can generate these visualizations directly from the code of a state machine.

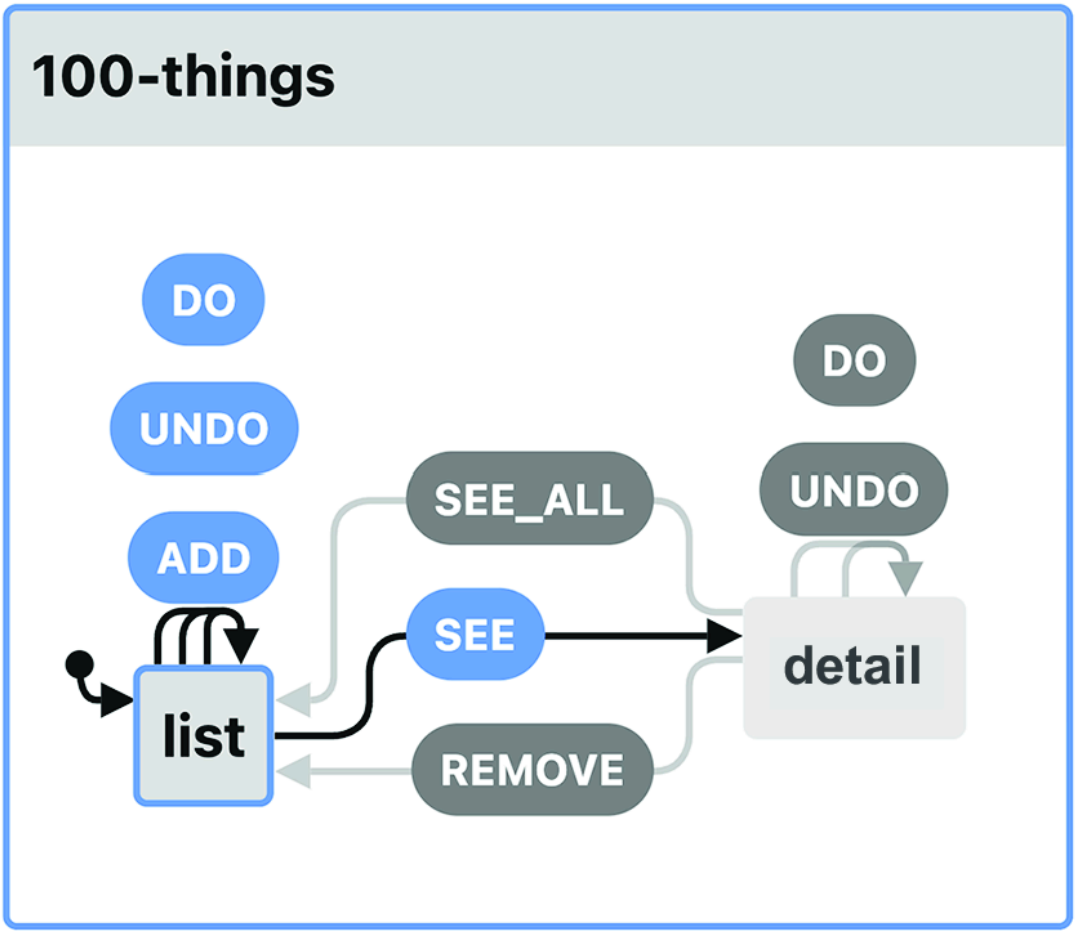

Figure 8.20 We've reduced the number of options a bit because we know how the application works. We can no longer add things in detail view or remove things in list view. This figure more closely represents the transitions in our application.

There's still something a bit weird about this machine, however. The `DO` action in list view is obviously different from the `DO` action in detail view. In detail view, we can do only that single thing, but in list view, we can do any of the things in the list. Let's differentiate these transitions as `DO_THIS` (detail view) and `DO_THAT` (list view), with the latter transition taking the `id` of the thing we're doing and vice versa for the `UNDO` transition to `UNDO_THAT` and `UNDO_THIS`. Figure 8.21 shows the result.

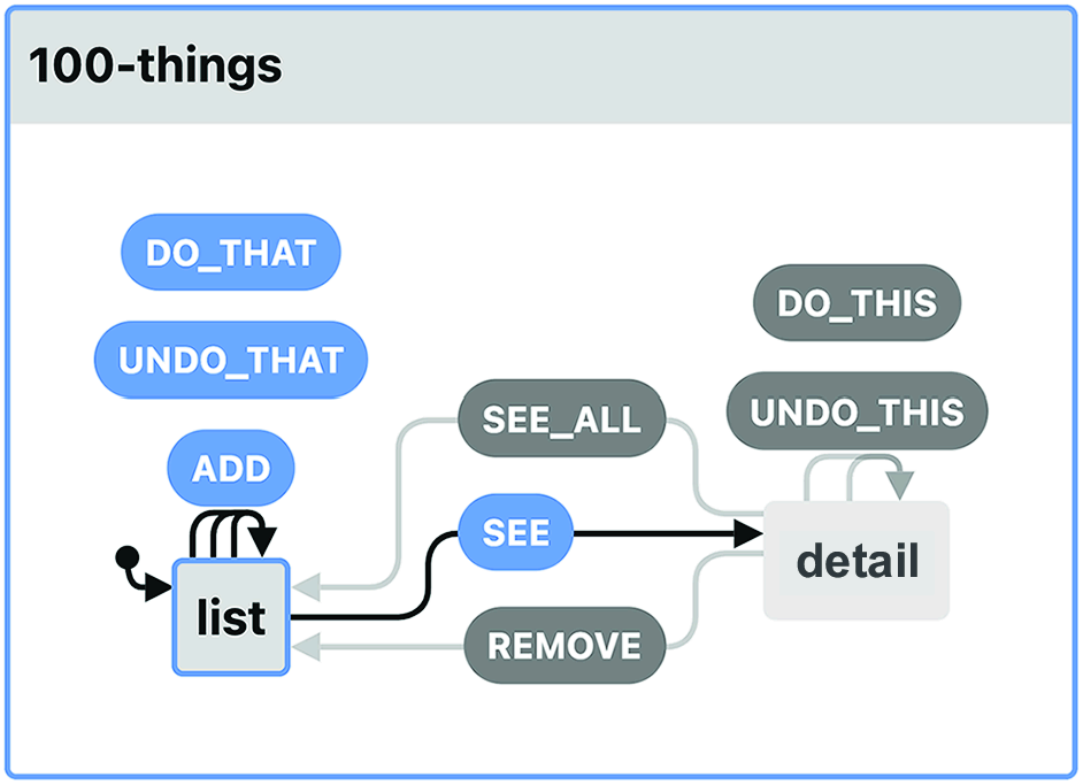

Figure 8.21 The new iteration of the state machine has different transitions for doing things in list view versus detail view.

The next step is defining the context data that exists above this state machine. This data consists of the same two values as earlier: the list of things and the current thing. When we do a transition, we can specify an action that is triggered when the transition occurs. When we want to see a thing and move from list view to detail view, we also need to specify which thing we're going to see, which we'll store in the current thing's context value. When we move from detail view to list view in the `SEE_ALL` transition, we must reset the current thing. Let's associate an action with each transition, as shown in figure 8.22; later, we'll define what these actions do.

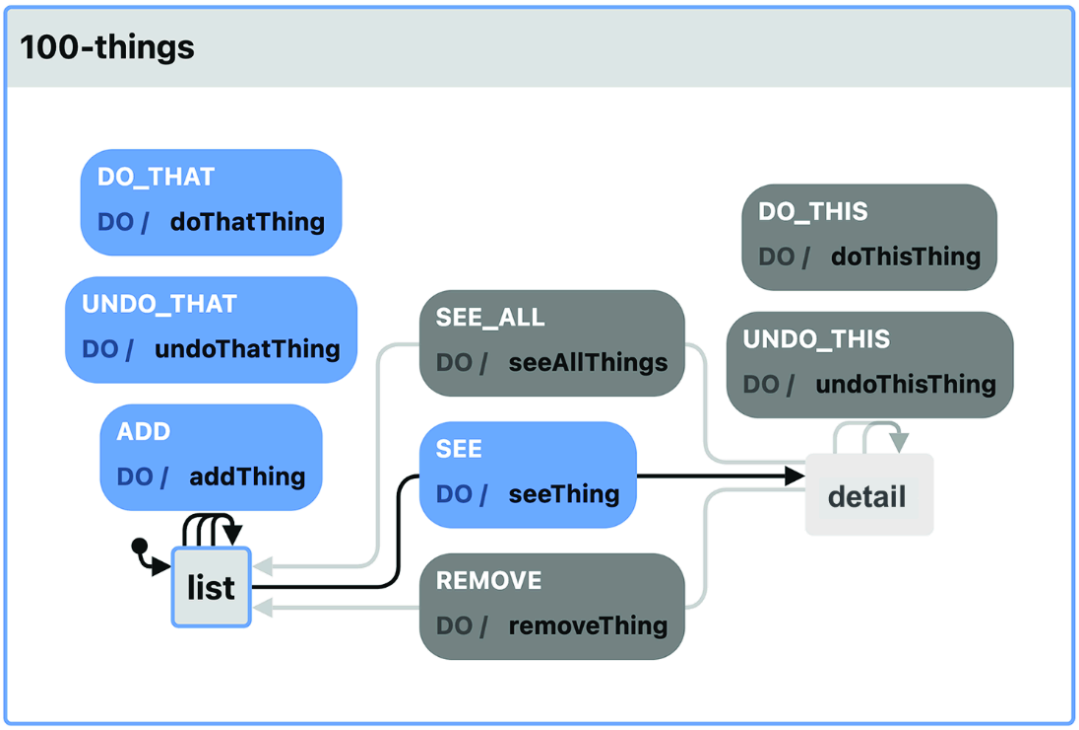

Figure 8.22 The state machine with actions annotated for each transition

We've implemented this final state machine along with the context and all actions in the application, but we've also implemented it in the stately.ai application to create the visualization in figure 8.22 (though the graphics might change over time). You can see this state machine and visualization at **https://mng.bz/aEOY**.

We're left with a slight problem. Remember that we used the same `useThing` hook in the `Thing` component in list view and the `SingleThing` component in detail view? Now we know that these two states are different and need different functions, so we're going to split the `useThing` hook into two different hooks instead. These hooks will be helpfully named `useThisThing` (used in `SingleThing` in single view) and `useThatThing` (used in `Thing` in list view).

How do we create this state machine? We'll discuss that topic when we get to the source code of the machine in section 8.7.2. Then we'll discuss how the actions are created in the source code for `actions.js`.

First, let's see how we can hook this machine up to our React codebase. We're still going to use a context, but this time, we're adding only a single static reference to our state machine. This value never changes, so the context never updates or causes re-renders. But the `useSelector`

hook from the XState library does cause updates when the machine changes, so this hook will fuel our application. We're also going to need to send transitions, which we'll do through our custom hook `useSend`. Now the application is structured as shown in figure 8.23.

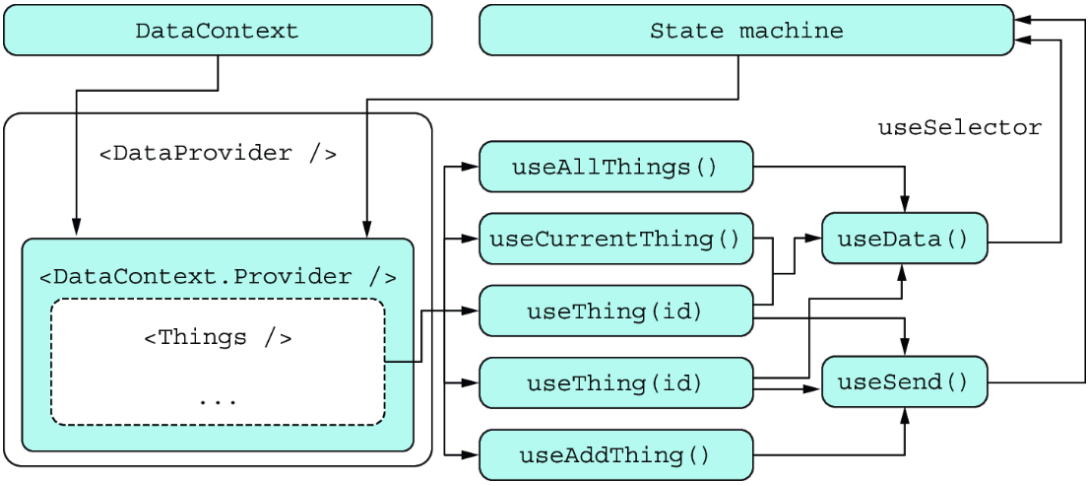

Figure 8.23 The architecture of our state machine–fueled application. Notice which custom hooks use data and send data. Four hooks use data, three hooks send data, and two hooks do both things.

8.7.2 Source code

For simplicity's sake, I split the machine into the definition of the machine itself and then the definition of the actions. I could have put everything in the same file, but my chosen approach makes the code look less daunting.

Persisting the data to local storage and reading it again happen in the data provider. We can conveniently override the initial state when we pass a machine to the `useInterpret` hook, and we can listen for any machine changes in a `useEffect` by using `service.subscribe`. Note that we store the whole state (which includes the context) in local storage, not the context alone. This approach is the correct way to restore a state machine in XState. It does mean that we'll put a lot more than the context in local storage, but that's not relevant for our focus.

Note that we're not using Immer-style reducers in this variant. We could, but we don't need to, as the reducers are fairly simple.

EXAMPLE: XSTATE

This example is in the `xstate` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch08/xstate
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch08-xstate**.

MACHINE.JS

This file is the meat of the application in this iteration. The machine governs how the application state advances, which transitions are allowed at any given state, and what happens when a translation occurs.

Listing 8.32 `src/data/machine.js`

```
import { createMachine } from "xstate";
import {
  doThatThing,
  undoThatThing,
  doThisThing,
  undoThisThing,
  removeThing,
  seeThing,
  seeAllThings,
  addThing,
} from "./actions";
export const machine = createMachine(    #1
  {
    predictableActionArguments: true,
    id: "100-things",    #2
    context: { things: [], currentThing: null },     #3
    initial: "list",    #3
    states: {
      list: {     #4
        on: {   #4
          DO_THAT: { target: "list", actions: "doThatThing" },   #4
          UNDO_THAT: { target: "list", actions: "undoThatThing" },  #4
          ADD: { target: "list", actions: "addThing" },   #4
          SEE: { target: "single", actions: "seeThing" },   #4
        },   #4
      },   #4
      single: {    #4
        on: {
          DO_THIS: { target: "single", actions: "doThisThing" },
          UNDO_THIS: { target: "single", actions: "undoThisThing" },
          REMOVE: { target: "list", actions: "removeThing" },
          SEE_ALL: { target: "list", actions: "seeAllThings" },
        },
      },
    },
  },
  {
    actions: {
      addThing,
      doThatThing,
      undoThatThing,
```

```
        doThisThing,
        undoThisThing,
        seeThing,
        seeAllThings,
        removeThing,
      },
    }
  );
```

#1 To create a new machine, we have to pass in configuration options.
#2 First, we have to provide a unique name. The name doesn't matter and can make debugging easier.
#3 We pass in an initial context and state, but we're going to overwrite the initial context later, so it doesn't matter here.
#4 Finally, we pass in all the states. For each state, we list which possible transitions can occur to which other state and with which accompanying action.

ACTIONS.JS

This file defines all the action creators that manipulate the state without using Immer, so we write them as immutable code and create minor utility functions to help. I could have used Immer, but this example reminds you that Immer is not mandatory and that our brains are still capable of doing things the old way. All the functions are straightforward, doing a single small thing each.

Listing 8.33 `src/data/actions.js`

```
import { assign } from "xstate";
import { v4 as uuid } from "uuid";
export const addThing = assign({
  things: (context, { name }) =>
    context.things.concat([{ id: uuid(), name, done: [] }]),
});
export const seeThing = assign({
  currentThing: (context, { id }) => id,
});
export const seeAllThings = assign({
  currentThing: null,
});
export const removeThing = assign({
  things: (context) =>
    context.things.filter((t) => t.id !== context.currentThing),
  currentThing: null,
});
function editThing(things, id, cb) {
  return things.map((t) =>
    t.id === id ? { ...t, done: cb(t.done) } : t
  );
}
function doSomeThing(things, id) {
  return editThing(things, id, (done) => done.concat(Date.now()));
}
export const doThatThing = assign({
  things: (context, { id }) => doSomeThing(context.things, id),
});
export const doThisThing = assign({
  things: (context) =>
    doSomeThing(context.things, context.currentThing),
});
function undoSomeThing(things, id, index) {
  return editThing(things, id, (done) =>
    done.slice(0, index).concat(done.slice(index + 1))
  );
}
export const undoThatThing = assign({
  things: (context, { id, index }) =>
    undoSomeThing(context.things, id, index),
});
```

```
export const undoThisThing = assign({
  things: (context, { index }) =>
    undoSomeThing(context.things, context.currentThing, index),
});
```

DATAPROVIDER.JSX

In this iteration, the data provider is merely a wrapper for the underlying state machine that does all the hard work, so the data provider is slim this time around.

Listing 8.34 `src/data/DataProvider.jsx`

```
import { useEffect } from "react";
import { useInterpret } from "@xstate/react";
import { machine } from "./machine";
import { DataContext } from "./DataContext";
const STORAGE_KEY = "100-things-xstate";
export function DataProvider({ children }) {
  const state =    #1
    JSON.parse(   #1
      localStorage.getItem(STORAGE_KEY)    #1
    ) || machine.initialState;    #1
  const service = useInterpret(machine, { state });
  useEffect(() => {
    const subscription =     #2
     service.subscribe((state) =>  #2
        localStorage.setItem(    #2
          STORAGE_KEY,    #2
          JSON.stringify(state),   #2
        )   #2
    );    #2
    return subscription.unsubscribe;   #2
  }, [service]);
  return (
    <DataContext.Provider value={service}>    #3
      {children}
    </DataContext.Provider>
  );
}
```

#1 The two primary responsibilities of the provider in this instance are to make sure that the initial state is set from local storage (or default to the initial state stored in the machine itself) . . .
#2 . . . and to store the current state in local storage when something changes in the machine.
#3 We stuff the machine inside a context only to get a reference to the machine later. The machine is a stable variable and will never change.

USEDATA.JS

When we need to display some data, we have to retrieve it from the machine context by using a selector function. But first, we have to get a reference to the machine from the data context.

Listing 8.35 `src/data/useData.js`

```
import { useContext } from "react";
import { DataContext } from "./DataContext";
import { useSelector } from "@xstate/react";
export function useData(selector) {
  const service = useContext(DataContext);
  return useSelector(service, selector);     #1
}
```

#1 To retrieve values from the machine (be it state or context values), we use a selector function from the library and apply that function to the machine from the context.

USESEND.JS

To manipulate the machine, we need to call the `send` method from the machine, as retrieved from the data context.

Listing 8.36 `src/data/useSend.js`

```
import { useContext } from "react";
import { DataContext } from "./DataContext";
export function useSend() {
  return useContext(DataContext).send;    #1
}
```

#1 To trigger transitions in the machine, we use the send method of the machine.

USEALLTHINGS.JS

This file retrieves data from the machine context by using the `useData` hook.

Listing 8.37 `src/data/useAllThings.js`

```
import { useData } from "./useData";
export function useAllThings() {
  return useData((state) => state.context.things)    #1
    .map(({ id }) => id);
}
```

#1 When we want to read data from the context, we go through the state.context property.

USETHATTHING.JS

This file is the first of the more complex hooks that both access data inside the state machine and allow you to manipulate the machine state by triggering various transitions.

Listing 8.38 `src/data/useThatThing.js`

```
import { useData } from "./useData";
import { useSend } from "./useSend";
export function useThatThing(id) {    #1
  const send = useSend();
  const thing = useData(({ context }) =>
    context.things.find((t) => t.id === id)
  );
  return {
    thing,
    seeThing: () => send({ type: "SEE", id }),
    doThing: () => send({ type: "DO_THAT", id }),
    undoLastThing: () =>
      send({ type: "UNDO_THAT", id, index: thing.done.length - 1 }),
  };
}
```

#1 The hook for a thing item in the list of all the things. It takes an id to identify which thing it works on.

USETHISTHING.JS

This hook also allows you to trigger various transitions, like the previous hook, but not the same transitions.

Listing 8.39 `src/data/useThisThing.js`

```
import { useData } from "./useData";
import { useSend } from "./useSend";
export function useThisThing() {    #1
  const send = useSend();
  const thing = useData(({ context }) =>
    context.things.find((t) => t.id === context.currentThing)
  );
  return {
    thing,
    removeThing: () => send("REMOVE"),     #2
    doThing: () => send("DO_THIS"),   #2
    seeAllThings: () => send("SEE_ALL"),    #2
    undoThing: (index) => send({ type: "UNDO_THIS", index }),
    undoLastThing: () =>
      send({ type: "UNDO_THIS", index: thing.done.length - 1 }),
  };
}
```

#1 This hook is used for a single thing, which is stored in currentThing, so we don't need to pass an id to it.
#2 Note that we can send a transition without properties as send("TYPE") rather than the more elaborate send({type:"TYPE"}).

USEADDTHING.JS

This hook allows you to create a single specific transition in the machine.

Listing 8.40 `src/data/useAddThing.js`

```
import { useSend } from "./useSend";
export function useAddThing() {
  const send = useSend();
  return (name) => send({ type: "ADD", name });
}
```

USECURRENTTHING.JS

This file is another trivial hook for accessing data from the machine context.

Listing 8.41 `src/data/useCurrentThing.js`

```
import { useData } from "./useData";
export function useCurrentThing() {
  return useData(({ context }) => context.currentThing);
}
```

INDEX.JS

We expose one more hook in this version.

Listing 8.42 `src/data/index.js`

```
export { DataProvider } from "./DataProvider";
export { useAddThing } from "./useAddThing";
export { useAllThings } from "./useAllThings";
export { useThatThing } from "./useThatThing";     #1
export { useThisThing } from "./useThisThing";   #1
export { useCurrentThing } from "./useCurrentThing";
```

#1 We have two hooks here, so we need to export both, and we also need to update both components using the old useThing hook, to use the relevant variant.

8.8 Data management recap

We've seen five ways to implement data management in this chapter, and we've hardly scratched the surface of what's available. We've implemented data management by using only pure React in section 8.3, and we've implemented it by using a few utility libraries, keeping pure React as the main data management library in section 8.4. The three external libraries we tried are Redux (Toolkit), zustand, and XState. Other notable alternatives include Jotai, Recoil, and Valtio. Refer back to chapter 1, figure 1.4, for an overview.

Although all these methods are different, they're all great. I would never fault anyone for using any of these tools. Some shine better in some circumstances than others, but they're all capable.

Often, you'll find yourself working on a project that uses one of these approaches, and it will probably be good enough. If you need to start a new project and have to choose, go with whatever is most comfortable for you. If you're in the rare situation of being able to choose *and* having the time and energy to learn something new, play with a few alternatives while mocking up the central parts of your application. You'll probably find a good candidate for your use case.

Finally, as demonstrated in this chapter, remember that your decision can always be reversed. With little effort, we were able to change the underlying data management library to something similar to or even far from the original while leaving almost all the presentation code intact. Don't be afraid to try something new.

Summary

- An application without data is meaningless. When we have a lot of data, especially if we need to manipulate it in complex ways, we need to think carefully about how to manage that data.
- Many excellent libraries for data management are available, but you don't need one unless you have a special circumstance. React's built-in functionality with context + `useReducer` will take you far, especially if you combine it with `useContextSelector` to minimize rendering.
- Redux is one of the most used data management libraries, mostly because it existed before React had good built-in capabilities itself, but also because it has been battle tested for ages and simply works. It has a bit of legacy boilerplate, however, even if you use RTK.
- Zustand is one of the newest and most popular kids on the data management block. It's so simple to use that you can't screw it up. It's a bit trickier to use at scale, however, because it doesn't come with any built-in tools or patterns for organizing multiple unrelated data stores.
- State machines have been all the rage for many years in many aspects of computer science, and you'll see them used in all sorts of programming languages, probably more so than in JavaScript. XState is a good library for JavaScript, though, and combined with React, it can improve your understanding of a given complex flow and re-

duce bugs because it forces you to think more deeply about what can happen when.

- When it comes to choosing the right tool for the right job, go with what you feel most comfortable with. All these tools and many more are essentially interchangeable with only minor changes to the code (as you can see in the implementations in this chapter), so even if you choose wrong at first, you can always change your mind later.