9 Remote data and reactive caching

This chapter covers

- Adding remote data to a React application
- Introducing the reactive cache
- Exploring remote data management libraries

All the applications we've built so far are local applications. By that, I mean applications that show data local to your computer. If you spin up one of the applications (such as the goal-tracking application from chapter 8), add some things to do, and then go to another computer, you cannot see the same data in any way. Even if you deploy the application to a website and go to the same website but on a different device (perhaps your phone), you will not be able to work on the same data that you do on your computer. You will also experience this situation if you use a different browser on the same computer.

The reason is that we store the data locally in the browser's local storage. Local storage is like a cookie but allows for more data in a more complex structure, and it's never sent to the server (as cookies can be).

That makes our applications far different from most applications you see out in the wild. On X (formerly Twitter), for example, we all see the same posts; you can interact with tweets by other people and they can interact with yours. This process requires sending the data to some remote server and receiving data from there.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch09**.

9.1 Server complexity

To make an application accessible to multiple users, you need to introduce server communication. When your computer is communicating

with a server, data is sent from your computer to the server in a request, and data is returned from the server as a response. When data moves from one computer to another, several complications are introduced, with one of the primary ones being latency. When you're using local data, that data is available in milliseconds, but when you're communicating with a server, network latency can cause data requests to take seconds. Compare the differences in figure 9.1.

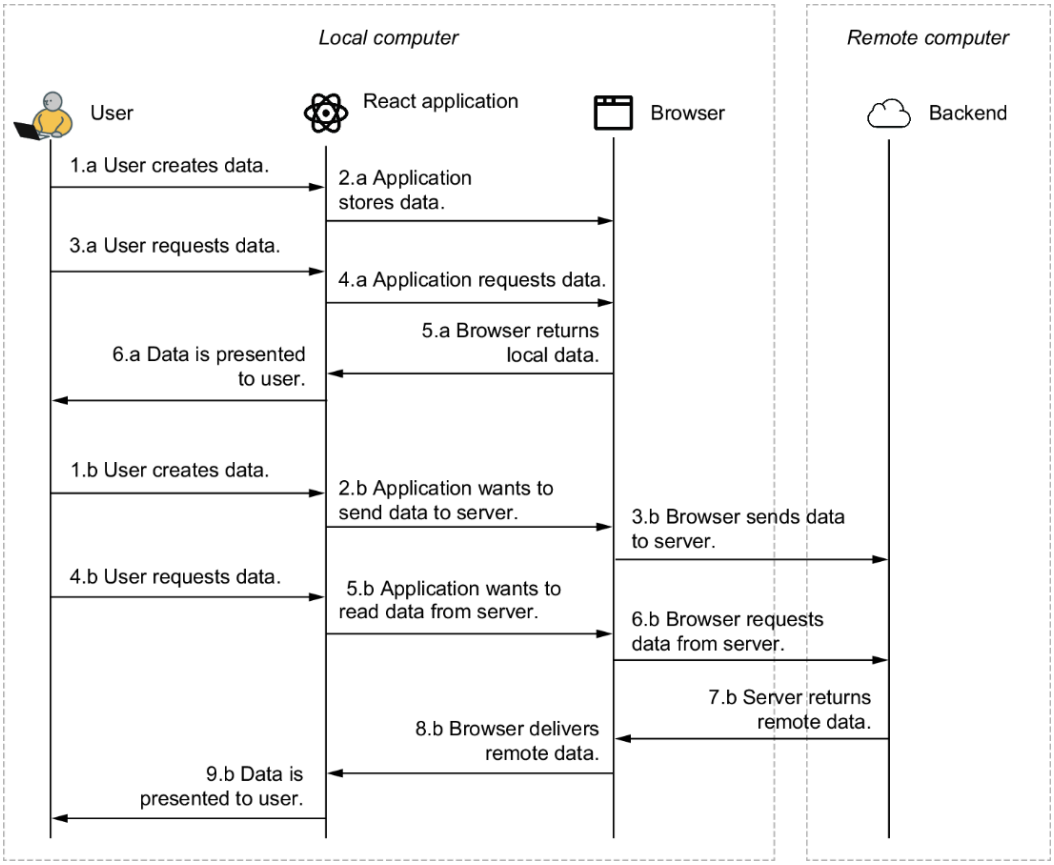

Figure 9.1 In the local-only scenario (1.a through 6.a), we use only data stored on the local computer. In scenario 1.b through 9.b, we go through a remote server, which introduces some extra delay and potential security problems but allows for collaboration between users and devices.

Adding remote data comes with security problems. When you are on X, only you can create and delete *your* tweets, so security is required to make sure that only you have access. This security requires logins, passwords, password recovery, two-factor authentication, social login (single sign-on), and a mess of problems, all of which don't involve React as much as they do the backend. We also need to worry about storing data efficiently in a database, handling multiple clients updating the same data, and so on. All these problems are well-known, nontrivial, backend problems.

Although I would love to introduce a backend framework to you (Express in Node.js and Django in Python are my favorites), that topic is beyond the scope of this book. You can find plenty of great books on those topics. Instead, I'm going to cheat a bit. We'll pretend that we interact with a remote server but the data is still local data. From the perspective of the React application, however, this data will for all intents and purposes work like true remote data. So we can create a local "server" that will never be exposed to the public; thus, we have no need for databases or security. See how this extra option works in figure 9.2.

WAIT—WE DON'T NEED A SERVER?

No. Creating a server is doable, of course, and we could even run a local server written in Node.js, Python, or some other backend language, but we don't need to. If we did, we would need to introduce a database, security measures, and more, which aren't necessary for React applications. Instead, to make things easy for ourselves, we are going to use a server mocking library called Mock Service Worker (MSW for short).

In this chapter, we will be using MSW for our backends, but the source code for those backends is available only in the online repository, as it is a bit too long and irrelevant to show here.

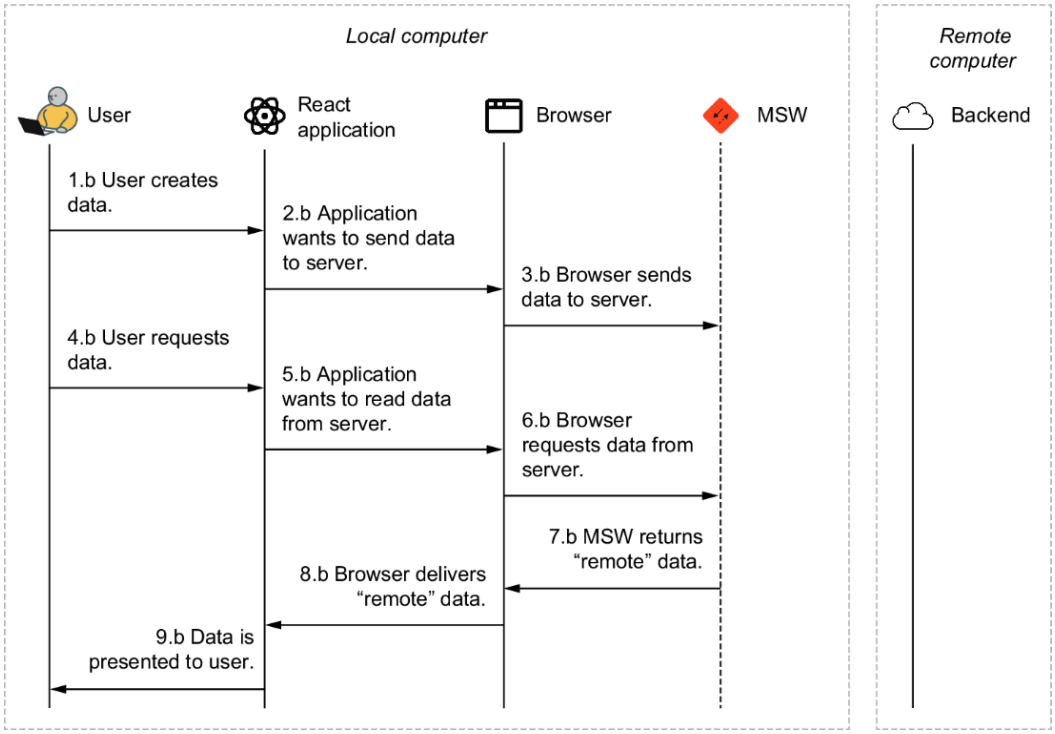

Figure 9.2 With our fake MSW (Mock Service Worker) intercept backend, React (or any other web application) will think that it's interacting with a real, live, web server when it's interacting with our middleware. We can create a React application as though we have a real backend without building said backend; we don't have to worry about databases, security, or anything like that.

In this chapter, we will start reexamining the goal-tracking application from chapter 8 using only the context, but this time with a web server handling all the requests. To make the application seem more real, we will add a login/signup screen, but because we use a fake web server, it won't validate the login; it simply allows you in right away regardless of how you log in.

To simulate a real web experience, we will introduce a small but noticeable lag in server responses. The initial simple solution isn't great because of the noticeable delay when nothing happens, but it works. Later, we will discuss how a reactive cache might be a much better solution and eventually reimplement the whole application by using TanStack Query, a reactive cache, data management library for React. We'll start by implementing the tool with the default bare-bones setup. Finally, we'll introduce client-caching principles that make the server-dependent experience much smoother for users.

9.2 Adding a remote server to do goal tracking

In this section, we'll add remote data handling to the goal-tracking application. To do so, we'll follow these steps:

1. Add a signup/login flow.
2. Design an API for our requests to go through.
3. Rewrite the data layer to use this new API.
4. Add a loading indicator.
5. Put everything together in the final application.

9.2.1 Adding signup and login

We need to add a login/signup screen. This screen is required because when we have a multitenant system (which lets multiple users sign in and work on their own data), we need to add some basic security. Alternatively, we could create a single-tenant system in which all users work on the same data, but that approach seems a bit weird for this type of application. We're not all working on the same goals, are we? Also, some troll would quickly delete all the data and/or add something obnoxious. Figure 9.3 shows the signup/login screen.

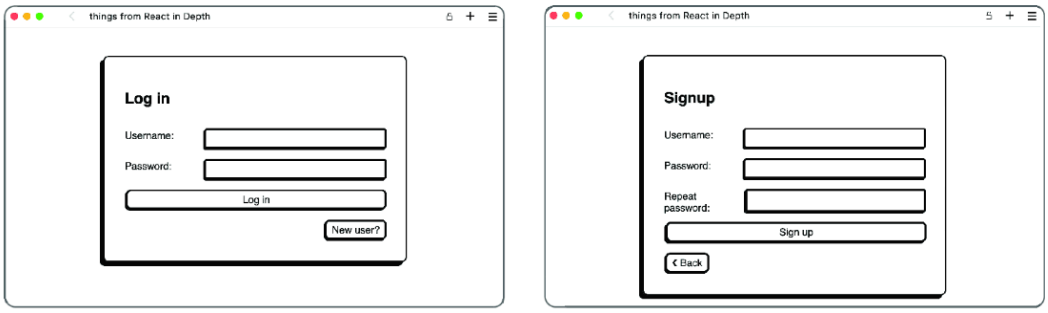

Figure 9.3 When you arrive at the new goal-tracking website for the first time, you have to log in or sign up. Ideally, we'd include a "forgot password" flow, but because we don't have a backend, we don't need that feature for now.

We'll also need to create new data management utilities for this authorization step as well as a new client UI. Finally, we need a logout button for the main list display.

9.2.2 Designing an API

In chapter 8, we created our goal-tracking application. This application handles retrieving data by looking inside our locally stored data, and it

handles writing data by updating that same locally stored data.
In this section, we are going to use that same architecture but reroute all the requests to a backend server. Well, we'll reroute them to MSW, but that fact is beside the point here. Whenever we want to retrieve data, we ask the server for the data, and whenever we want to write data, we send the new data to the server and receive updated data. Compare these approaches in figure 9.4.

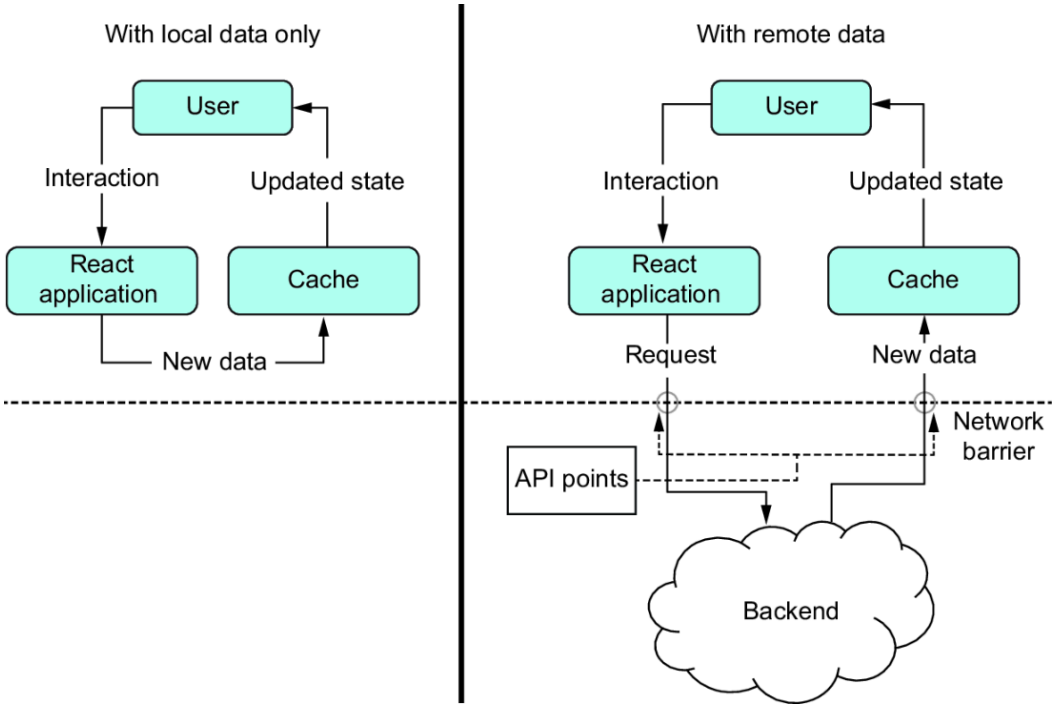

Figure 9.4 When we store data locally, we can dip into the local data when we want to display it, but when we store the data remotely, we have to wait for the server to respond before we can display anything. This process has to occur every time we read or write data. Note also that every time we move from local to remote data, we have to agree on the format between the two parts of the applications; this format agreement is known as the API.

Note the interface between the application and the web server, which is known as an *application programming interface* (API). API design is a category of its own, but for this application, we're going to create a RESTful CRUD interface.

RESTFUL CRUD?

Both of these terms are related to API design, which is beyond the scope of this book.

REST (Representational State Transfer) can be boiled down to requests being stateless (thus, self-contained) and resource centric. In a RESTful API, you don't have a login method; you have a session resource that you create by sending a username and a password.

CRUD is an acronym for *create*, *read* (or *retrieve*), *update*, and *delete*—the four basic operations that you perform on a resource in a system such as this one. So for every type of resource, you can create a resource, list (read) some or all resources, update a resource, or delete a resource. Not all operations make sense for all types of resources, of course. You can even have nested resources, such as songs on a certain album or books by a certain author.

TIP If you want to learn more about API design, check out *API Design Patterns*, by JJ Geewax, at **https://www.manning.com/books/api-design-patterns**.

With these principles in mind, we have the following endpoints in our backend:

- `POST /api/session`—Log in to the application.
- `POST /api/user`—Create a new user.
- `DELETE /api/session`—Log out of the application.
- `GET /api/user`—Get the current user.
- `GET /api/things`—List all things by the current user.
- `GET /api/things/<id>`—Get all information about the thing with the given `id`.
- `DELETE /api/things/<id>`—Delete the given thing.
- `POST /api/things`—Create a new thing.
- `DELETE /api/things/<id>/done/<id>`—Delete the specific time we did the specific thing.
- `DELETE /api/things/<id>/done/last`—Delete the most recent time we did the specific thing.
- `POST /api/things/<id>/done`—Do this specific thing.

With these endpoints, we can replicate all the things we can already do in our application. Note that we haven't defined any API endpoints for updating resources; there isn't anything to update anywhere in the application. We don't even support renaming things. We can only *do* or *undo* things (which in API terms is creating and deleting "done" resources), not *update* the thing itself.

This book doesn't cover the reasoning behind the design and structure of this API. Suppose that a backend coworker tells you that you have to work with this API, and you trust them to have made it correctly. This coworker seems like a nice, competent person, though they do have a weird mania about painting plastic miniatures for board games, but that topic is also beyond the scope of this book.

9.2.3 Rewriting the data layer

Single source of truth is an important concept in computer science, and this principle comes into play here. We store the data in the database on the server, and that server is our single source of truth. Thus, we cannot also store the data elsewhere because that might lead to a conflict if the two sources differ on some data.

Due to this principle, we must hold the data in the backend as the only true source of content. Theoretically, multiple people could be working on the same data, so just because you delete a thing and go back to the list of all things, you cannot assume that only that single change was made. Multiple other changes could have been made. So you have to query the server every time you want to display the list for the data to be considered "live."

Using this principle, we need to query the server every time we update content. We cannot be completely sure what our changes did to the content (the server might reject creating a thing with certain words, for example), and other users might be interacting with the same content. This situation sometimes results in two requests being required for an operation, resulting in double network latency. All these requirements lead to the flow of data shown in figure 9.5.

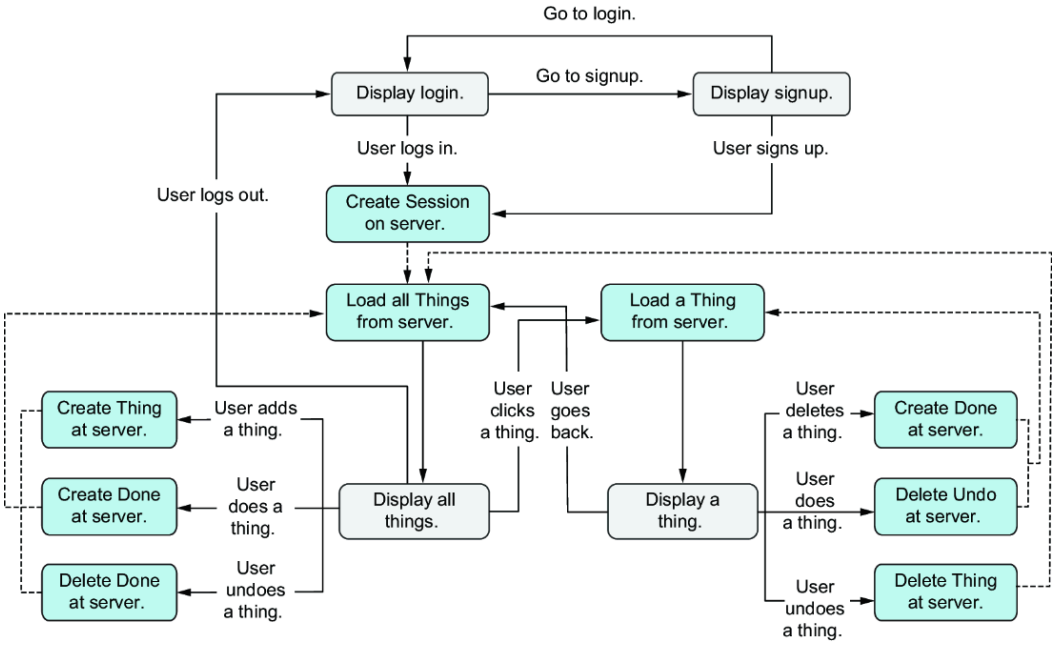

Figure 9.5 We have to request updated data from the server every time we want to display a page to make sure that the data displayed is current. The light-gray boxes are the user screens in the application, and the gray boxes are the server requests. The dashed arrows are the forced updates for new data that must happen after other requests.

Redesigning our entire data layer around this data flow means that we can still use the same structure of the four primary hooks and the global context to access and manipulate data, but behind the scenes, they will do very different things. Before we get too far, we also need to split one of the hooks, `useThing`, into two different hooks, `useThatThing` and `useThisThing`, as we did in the XState example in chapter 8. On top of all that, we need new hooks for the login/signup view and for the logout button. Figure 9.6 shows this rework.

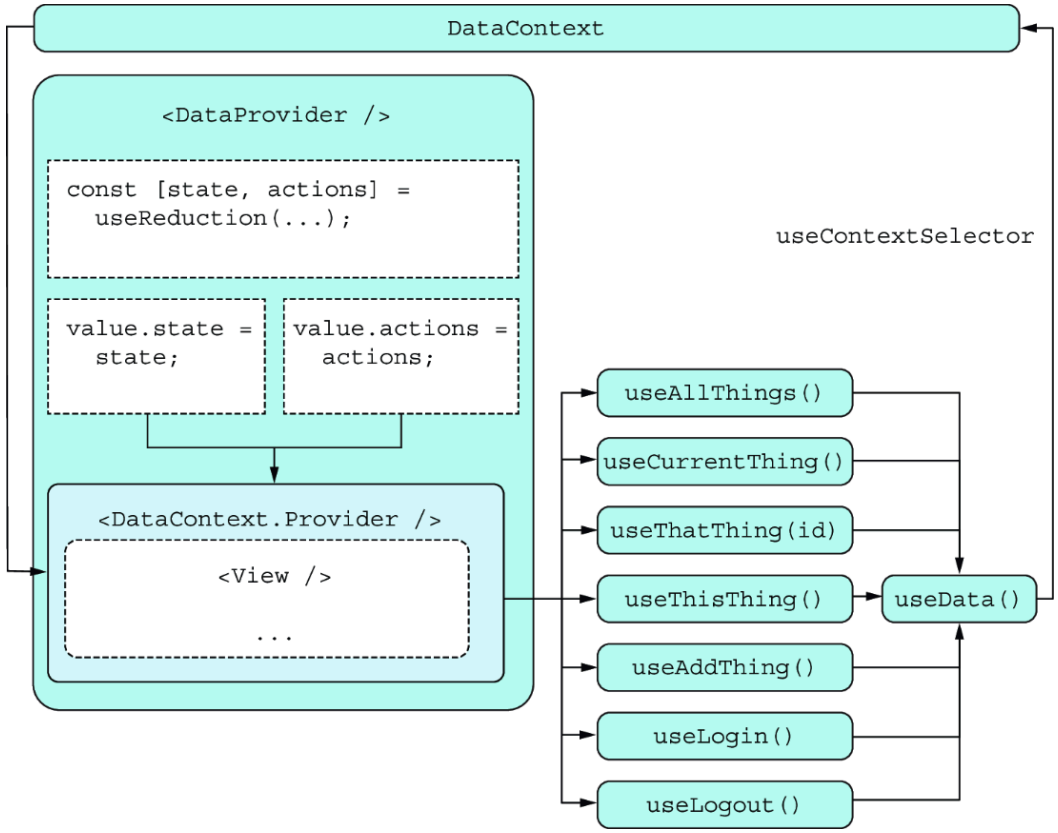

Figure 9.6 Now we require seven primary hooks through which the main application can interact with the data layer.

As a final note on the data layer, we are making some changes to what data is displayed where. Most important, we're adding a new description field to each thing that will be displayed in the thing's detail view but not in the list view. This pattern is fairly common: less information is used in list view (which might be a table of employees) than in detail view (which might be full information about each employee), and it allows us to discuss some important data management topics later.

You can see the updated form for adding a new thing as well as the detail view in figure 9.7. The new description field is displayed right below the thing name.

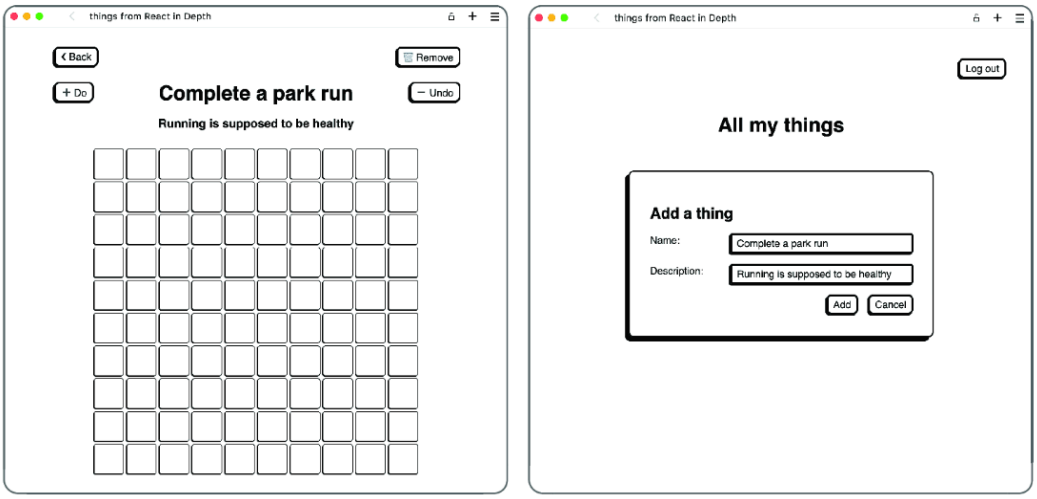

Figure 9.7 Things now have a more detailed description that will be visible only in detail view for a single thing (*Running is supposed to be healthy* on the right side of the figure). The form for adding a thing also looks more like the forms for logging in and signing up, ensuring visual consistency throughout the application. We have a logout button, too!

9.2.4 Adding a loading indicator

Because we now have a server that has to respond to most user actions, we also start to have noticeable lag. We will have double lag in many operations, as I explained in section 9.1.3. User experience (UX) research shows that an application that reacts within 100 milliseconds (ms) is said to react instantly and that no further indication of wait time should be required. Delays longer than 100 ms require some form of loading indicator so the user won't be furious about the lack of feedback and start clicking buttons twice, refreshing the page, or similar actions.

In this section, we will introduce a global loader to the application, overlaying the entire interface whenever some load is happening. This loader will look like figure 9.8—but animated, of course. (Why aren't GIFs supported in print media?)

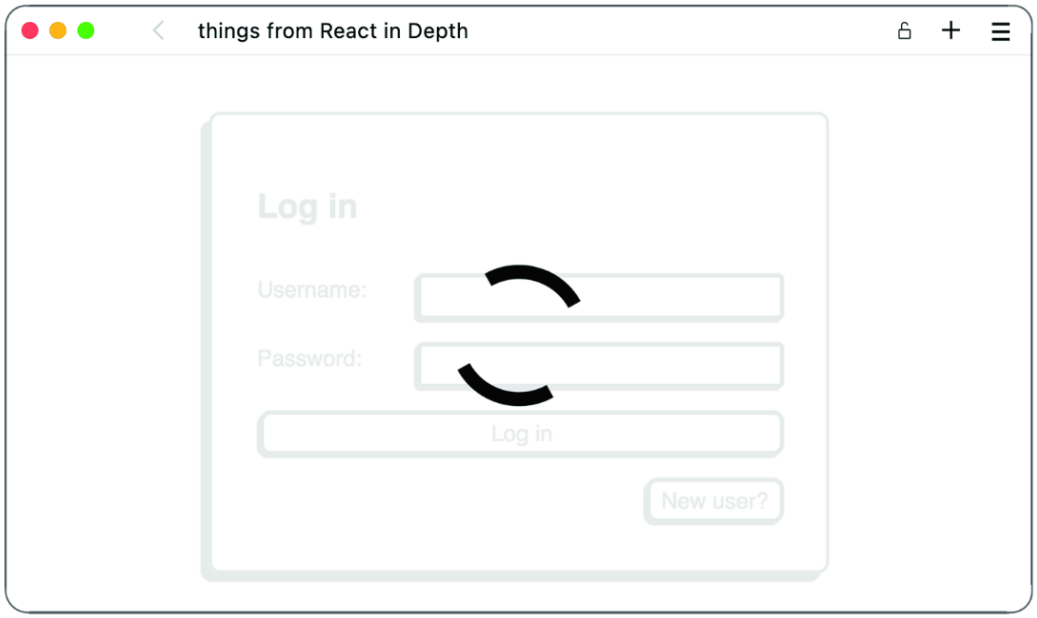

Figure 9.8 A simple loader with two spinning quarter-circles. The background is dimmed 90% while the loader is displayed.

To achieve this effect technically, we'll add a stateful Boolean, `isLoading`, that flips correctly when any server request starts and completes, respectively.

9.2.5 Putting everything together

The full application is getting large at this point, so I'm not going to show the entire source code. The application exceeds 1,000 lines of code, which would take up quite a few pages. I will highlight the most important bits of code in listings in the section, however. You can find the full source code in the online repository.

COMPONENT HIERARCHY

Let's start with the full component hierarchy diagram, which contains a lot more than it did earlier due to the new authorization flow (figure 9.9).

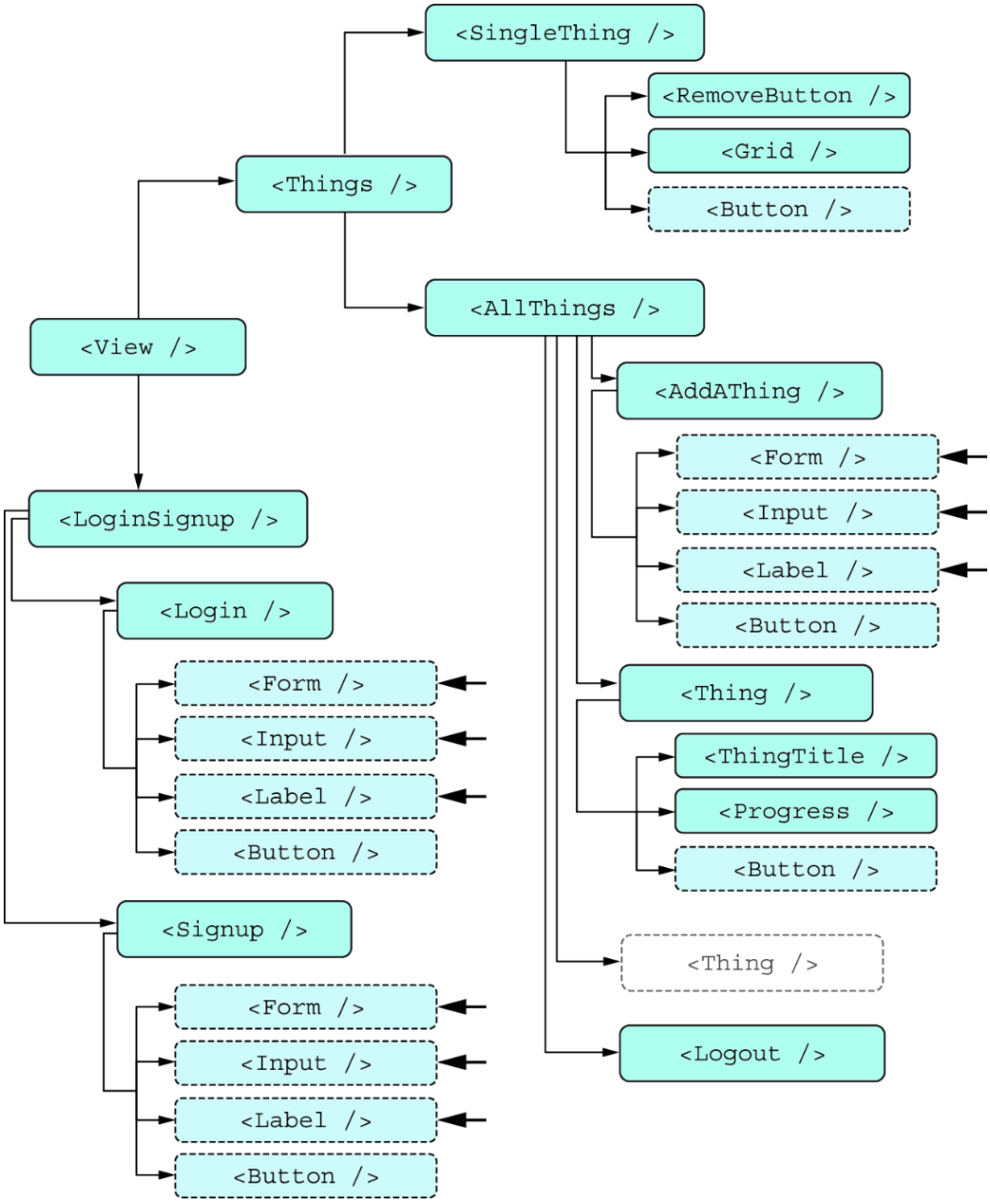

Figure 9.9 The full component hierarchy diagram. The dashed components are the simpler UI-only components that we use to generalize the UI and reuse functionality. Notice that besides a reusable button, we have a form, an input field, and a label, as marked by the short, fat arrows.

FILE STRUCTURE

The complete file structure for this application is a bit comprehensive, with a whopping 38 files and more than 1,000 lines of code, which is why I can't show all the source code in this chapter. This snippet lists the full folder and file structure:

```
src/
  backend/
    index.js    #1
  data/
    DataContext.js     #2
    DataProvider.jsx   #3
    index.js    #2
    Loader.jsx    #4
    useAddThing.js   #2
    useAllThings.js   #3
    useAPI.js   #3
    useCurrentThing.js   #2
    useData.js    #2
    useLoginSignup.js   #5
    useLogout.js    #5
    useThatThing.js   #3
    useThisThing.js   #3
    useUser.js   #5
  view/
    AddAThing.jsx   #2
    AllThings.jsx    #2
    Button.jsx    #2
    Form.jsx   #5
    Grid.jsx    #2
    index.js    #2
    Login.jsx    #5
    LoginSignup.jsx   #5
    Logout.jsx    #5
    Progress.jsx    #2
    RemoveButton.jsx  #2
    Signup.jsx    #5
    SingleThing.jsx   #2
    Thing.jsx   #2
    Things.jsx    #2
    ThingTitle.jsx    #2
    View.jsx     #6
  App.jsx    #2
  main.jsx  #2
```

#1 The fake intercept backend created with MSW

#2 All these files are mostly identical with the previous versions (with only minor changes where relevant).

#3 The new main server integration takes place in these files, which are included in full in this section.

#4 The new loader I alluded to, which is mostly CSS

#5 Because we needed authorization, we have a bunch of files related to displaying login and signup forms as well as to handling the events.
#6 The new main component that switches between the login/signup flow (unauthorized access) and the main application (authorized access)

IMPORTANT CODE SNIPPETS

This section discusses the most important code snippets, which are mostly about all the data bits, of course. In particular, we'll discuss the data provider and each individual data hook that feeds data and interaction into the application.

Let's start with a new hook called `useAPI`. This hook exposes several low-level network requests wrapped in a simple function that toggles the loading flag correctly. The user functions also trigger the authorization flag. Also, both of said flags are exported. The following listing shows this file.

Listing 9.1 `src/data/useAPI.js`

```
import { useState, useMemo } from "react";
const URLS = {
  SESSION: "/api/session",    #1
  USER: "/api/user",    #1
  THINGS: "/api/things",  #1
  THING: (id) => `/api/things/${id}`,   #1
  DONES: (id) => `/api/things/${id}/done`,   #1
  DONE: (id, did) =>   #1
    `/api/things/${id}/done/${did}`,   #1
};
const get = (url) => fetch(url);     #2
const post = (url, data = null) => fetch(   #2
  url,   #2
  { method: "POST", body: JSON.stringify(data) }    #2
);    #2
const remove = (url) =>    #2
  fetch(url, { method: "DELETE" });    #2
export function useAPI() {
  const [isAuthorized, setAuthorized] =   #3
    useState(false);    #3
  const [isLoading, setLoading] =     #4
    useState(false);    #4
  const API = useMemo(() => {
    const auth = () => setAuthorized(true);     #5
    const unauth = () => setAuthorized(false);    #5
    // common wrapper for all network requests
    const wrap = (promise) => {     #6
      setLoading(true);
      return (
        promise
          // This is invoked regardless of result
          .finally(() => setLoading(false))
          // This is invoked for all server responses,
          // but .ok is only true for 20x and 30x responses
          .then((res) => {
            if (!res.ok) {
              throw new Error("Unauthorized");
            }
            return res.json();
          })
          // Finally, this catches both network, auth, and server errors
          .catch((e) => {
            unauth();
            throw e;
```

```
          })
      );
    };
    // USER API
    const getUser = () =>     #7
      wrap(get(URLS.USER)).then(auth); #7
    const login = (data) =>   #7
      wrap(post(URLS.SESSION, data)).then(auth);    #7
    const signup = (data) =>   #7
      wrap(post(URLS.USER, data)).then(auth);    #7
    const logout = () =>  #7
      wrap(remove(URLS.SESSION)).then(unauth);    #7
    // THING API    #7
    const loadThings = () =>   #7
      wrap(get(URLS.THINGS));    #7
    const loadThing = (id) =>    #7
      wrap(get(URLS.THING(id)));   #7
    const addThing = (data) =>   #7
      wrap(post(URLS.THINGS, data));    #7
    const removeThing = (id) =>    #7
      wrap(remove(URLS.THING(id)));   #7
    const doThing = (id) =>    #7
      wrap(post(URLS.DONES(id)));   #7
    const undoThing = (id, did) =>    #7
      wrap(remove(URLS.DONE(id, did)));    #7
    return {
      getUser,
      login,
      signup,
      logout,
      loadThings,
      loadThing,
      addThing,
      removeThing,
      doThing,
      undoThing,
    };
  }, []);
  return { isAuthorized, isLoading, API };
}
```

#1 First is the URL structure used by the API, extracted to make it easier to update.

#2 Second, we have some simple wrappers for the built-in fetch() function to make GET, POST, and DELETE requests.

#3 When the user is logged in (or has recently signed up), the authorization flag is

true; otherwise, it's false.

#4 Whenever some request is ongoing, the loading flag is true. As soon as the request completes, whether or not it completes with an error, the flag is toggled to false.

#5 Two small utility functions for toggling the authorization flag

#6 The request wrapper function handles errors as well as toggles the loading flag.

#7 The primary part of this hook: the API requests. Note that we don't handle the responses here (except for the authorization functions); we merely make sure that they connect to the right place.

Next up is the data provider in `DataProvider.jsx`. This file is a more business-aware layer built on top of the API exposed in the previous `use-API` hook.

Listing 9.2 `src/data/DataProvider.jsx`

```
import { useState, useMemo } from "react";
import { DataContext } from "./DataContext";
import { Loader } from "./Loader";
import { useAPI } from "./useAPI";
export function DataProvider({ children }) {
  const { isAuthorized, isLoading, API } =    #1
    useAPI();    #1
  const [things, setThings] = useState([]);   #2
  const [currentId, setCurrentId] =     #3
    useState(null);   #3


  const [currentThing, setCurrentThing] =    #4
    useState(null);  #4
  const actions = useMemo(() => {
    const { getUser, signup, login, logout } =    #5
      API;    #5
    const loadThings = () =>     #6
      API.loadThings().then(setThings);   #6
    const loadThing = (id) =>   #6
      API.loadThing(id).then(setCurrentThing);    #6
    const addThing = (data) =>    #6
      API.addThing(data).then(loadThings);    #6
    const seeThing = (id) =>    #6
      setCurrentId(id);    #6
    const seeAllThings = () =>    #6
      setCurrentId(null);    #6
    const removeThing = (id) =>   #6
      API.removeThing(id).then(seeAllThings);   #6
    const doThisThing = (id) =>   #6
      API.doThing(id).then(loadThings);   #6
    const undoLastThing = (id) =>    #6
      API.undoThing(id, "last").then(loadThings);   #6
    const doThatThing = (id) =>   #6
      API.doThing(id).then(() => loadThing(id));    #6
    const undoThatThing = (id, did) =>    #6
      API.undoThing(id, did)   #6
        .then(() => loadThing(id));   #6
    return {
      addThing,
      doThatThing,
      doThisThing,
      getUser,
      loadThing,
```

```
        loadThings,
        login,
        logout,
        removeThing,
        seeAllThings,
        seeThing,
        signup,
        undoThatThing,
        undoThisThing,
      };
    }, [API]);
    const value = {
      state: { things, currentId, currentThing, isAuthorized },
      actions,
    };
    return (
      <DataContext.Provider value={value}>
        {isLoading && <Loader />}     #7
        {children}
      </DataContext.Provider>
    );
  }
```

#1 We extract the two flags and the API object from the useAPI hook.
#2 We create some local state values. This one is a temporary store for the list of things retrieved from the server. We never manipulate this value directly; we only store what the server tells us to store.
#3 The current id is the only part of the state we manipulate directly when we go from list view to detail view, and vice versa.
#4 The current thing is similarly loaded from the server only.
#5 Four of the API operations are used as they are. We don't need to add any functionality when a user logs in or signs up.
#6 Here, we create our main interaction functions, mostly by wrapping the API calls and either setting state (after data load) or loading fresh data (after data mutation).
#7 In the returned JSX, we include the loader only if the global loading flag is true.

When we interact with the data provider, we mostly retrieve functions or values directly, so most of the data hooks are super trivial. The hook for listing all the things is a bit more complex because it also initiates loading.

Listing 9.3 `src/data/useAllThings.js`

```
import { useEffect } from "react";
import { useData } from "./useData";
export function useAllThings() {
  const loadThings = useData(({ actions }) => actions.loadThings);
  useEffect(    #1
    () => void loadThings(),   #1
    [loadThings],   #1
  );   #1
  return useData(({ state }) => state.things)     #2
    .map(({ id }) => id);    #2
}
```

#1 When this hook is used, all the things are loaded from the server. void is a JavaScript keyword for throwing away a result and returning nothing. Remember that effect functions cannot return anything except a cleanup function, so using void is a trick to make sure that nothing is returned.

#2 When the results are in, we map it to a list of ids.

When we want to display a single thing inside the list of things, we need access to some of the data-mutation functions as well as to the single thing extracted from the full list.

Listing 9.4 `src/data/useThisThing.js`

```
import { useData } from "./useData";
export function useThisThing(id) {
  const thing = useData(({ state }) =>     #1
    state.things.find((t) => t.id === id)    #1
  );    #1
  const { seeThing, doThisThing, undoLastThing } = useData(
    ({ actions }) => actions
  );
  return {
    thing,
    seeThing: () => seeThing(id),     #2
    doThing: () => doThisThing(id),    #2
    undoLastThing: () => undoLastThing(id),   #2
  };
}
```

#1 A single thing is retrieved from the list of all things by the given id.

#2 Besides the data about the thing, we return some functions curried with the given id.

On the other hand, when we need to display the detail view for a single thing, we can't rely on the existing data in the state. We have to fetch that single thing from the server so that we have all the right information available. (Remember the thing description; we return that information only from the server in detail view, not list view.) This hook is implemented in the following listing.

Listing 9.5 `src/data/useThatThing.js`

```
import { useEffect } from "react";
import { useData } from "./useData";
export function useThatThing() {
  const id = useData(({ state }) =>     #1
    state.currentId);    #1
  const thing = useData(({ state }) =>  #1
    state.currentThing);   #1
  const {
    seeAllThings,
    doThatThing,
    undoThatThing,
    removeThing,
    loadThing
  } = useData(({ actions }) => actions);
  useEffect(    #2
    () => void loadThing(id),    #2
    [id, loadThing],   #2
  );   #2
  return {
    thing,
    removeThing: () => removeThing(id),
    doThing: () => doThatThing(id),
    seeAllThings,
    undoThing: (did) => undoThatThing(id, did),
    undoLastThing: () =>
      undoThatThing(id, "last"),
  };
}
```

#1 When we load the detail view of a thing, we need access to the thing we're focusing on.
#2 We have to remember to fetch all the details of the current thing based on the given id.

Finally, to know whether we should display the public login/signup flow or the restricted private data that belongs to the currently authorized

user, we need to retrieve the current user. If such a user exists, the authorization flag will update correctly.

Listing 9.6 `src/data/useAddThing.js`

```
import { useEffect } from "react";
import { useData } from "./useData";
export function useAddThing() {
  const getUser = useData(({ actions }) => actions.getUser);
  const isAuthorized = useData(({ state }) => state.isAuthorized);
  useEffect(() => void getUser(), [getUser]);    #1
  return isAuthorized;    #2
}
```

#1 When this hook is used, the user is loaded from the server, and if that load succeeds, the API sets the authorization flag.
#2 This hook returns the authorization flag, which is always false on the initial render but might later update to true if the request succeeds.

9.2.6 Evaluating the solution

This final application works. We've added a nice, functional login/signup flow, though for a production-grade system, we would probably need to add a password recovery flow as well. You can have a go at it yourself in the repository.

EXAMPLE: THINGS

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

```
$ npm run dev -w ch09/things
```

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/ch09-things**.

The flow of logic, which requires us to rerun the loaders every time we update something, can get quite complex. We could easily imagine many more ways of updating parts of a system, which would require even more connections between updaters and loaders. That process could become quite a problem. It's also a little confusing that we still have local state variables but must remember not to change them directly, as that would

violate the single-source-of-truth principle (and make the server state inconsistent).

To enable us to work better with remote data—including keeping a logical relationship between loading data and updating data and providing better cache management—other libraries have sprung up. One of these libraries is TanStack Query, which we'll look at in section 9.3. TanStack Query is a remote-first state management library that is excellent at doing these things, though it also has some shortcomings. We'll start by using the TanStack Query library with bare-bones capabilities; later, we'll add extra layers of logic for improved UX.

9.3 Migrating to TanStack Query

The implementation we created in section 9.1 works. It has some drawbacks, but all in all, it works, and it's a fine solution for many applications as well. You don't need any special libraries to handle an application powered by server data.

That said, we can optimize a few things. Although the application isn't small, it's not big, and many painful problems start to occur only at larger scale. We can see some of those problems starting to creep into even this minor application.

First, we have to store the remote data somewhere locally (that is, *cache* it), and we have to decide whether to read from the cache or fetch fresh information from the server. Furthermore, we have to fetch fresh information from the server manually when we perform various operations, even though we may not need that information currently; we do it only to be safe and make sure that the cache is fresh. Suppose that we increment the number of times we did a thing three times instantly. In the setup from section 9.1, we would be fetching the full list of all the things every time, rather than once at the end of all three increments.

A better way to handle such a cache setup is *cache invalidation*. When we operate on the remote data, it would be better simply to mark the local cached data as invalidated, which means that the next time we need this data, it's prudent to fetch new data from the server.

This is exactly what the TanStack Query library does. Among other things, the library introduces us to a common dichotomy of remote data han-

dling: queries versus mutations. *Queries* fetch data (*retrieve* data, the *R* in *CRUD*), whereas *mutations* change data (*create*, *update*, and *delete*, the remaining letters of *CRUD*).

TanStack Query is not the only library of its kind, of course, but it is the most popular. A rival library is SWR, a name derived from the stale-while-revalidate caching strategy. Stale-while-revalidate partially underlies the design of TanStack Query, as we will discuss. In some sense, SWR is a simpler library to use, but because TanStack Query is much more popular, we will use it in this chapter.

9.3.1 TanStack Query architecture

One important thing to understand about TanStack Query and many similar libraries is that it doesn't handle remote communication. It is merely a library for dealing with remote data caching in a sensible way.

To use the library, you have to define the functions that interact with the server yourself. These functions, however, are plain JavaScript functions that return a promise, which eventually resolves to the data requested (or performs the desired mutation). Fortunately, the wonderful built-in browser function `fetch()` can do that work for us.

If you want to use TanStack Query to get remote data from the path `"/api/ customers"`, you simply create a function that can do that, and later, you can pass that function to TanStack Query:

```
const getCustomers = () => fetch('/api/customers/')
  .then((res) => res.json());     #1
```

#1 We need this line to read the response as a JSON document.

If you need to create a mutation in TanStack Query that can send data as a `POST` request to the same path (to create a new customer), you'd create a second function that will take the new data as an argument. Note that for some reason, TanStack Query only allows functions to take a single argument, so make sure to bundle the data in a single object:

```
const createCustomer = (data) =>
  fetch("/api/customers/", {
    method: "POST",
    data: JSON.stringify(data),
  }).then((res) => res.json());
```

A second important bit of TanStack Query architecture is that it works around a single central cache where all the currently cached responses exist—a so-called *query client*. For the central query cache to work in your application, you have to create a new query client at the top level and wrap the entire thing in a `QueryClientProvider` with the given client. This process is reminiscent of how contexts work and essentially the same under the hood. The syntax is familiar:

```
import {     #1
  QueryClient,    #1
  QueryClientProvider,   #1
} from "@tanstack/react-query";  #1
const queryClient = new QueryClient();     #2
function App() {
  return (
    <QueryClientProvider client={queryClient}>     #3
      <TheInnerApplicationGoesHere />     #4
    </QueryClientProvider>
  );
}
export default App;
```

#1 Loads the relevant functions from the library
#2 Creates a query client outside the function, so it's created once as a global variable
#3 To make all hooks work, we provide the query client to the entire application.
#4 We include the main application inside this provider.

With this setup completed, we are ready to start using this library.

9.3.2 Queries and mutations

TanStack Query exposes two main hooks, through which you interact with the library on the surface. These hooks are `useQuery` for making queries and `useMutation` for—you guessed it—making mutations.

QUERYING

Queries are made with an existing function, as I explained earlier. You also need an identifier that represents this query: the query key. TanStack Query allows arrays of any combination of values to serve as query keys, but it is common to make the query key an array with a single string (if the query is simple and flat) or an array with a string first and an object second. Let's see some examples. If you make a query for the list of all customers, you would use the array `["customers"]` as the query key:

```
const response = useQuery({
  queryKey: ["customers"],
  queryFn: getCustomers,
});
```

But if you read only a single customer or the first page of customers or maybe search for a specific customer, you would need to include any such parameter as part of the query key:

```
// Get the customer with ID=1
const customerOne = useQuery({
  queryKey: ["customer", { id: 1 }],
  queryFn: () => getCustomer(1),
});
//
// Get page 3 of all customers
const customersP3 = useQuery({
  queryKey: ["customers", { page: 3 }],
  queryFn: () => getCustomersByPage(3),
});
//
// Get customers named "John"
const johns = useQuery({
  queryKey: ["customers", { name: "John" }],
  queryFn: () => getCustomersByName("John"),
});
```

Note that you would have to create these underlying functions (`getCustomers` , `getCustomer` , `getCustomersByPage` , and `getCustomersByName`) yourself and make sure that they passed the given arguments to the server. The smart thing about these query keys is that we can later reference queries by the exact key or by a partial key. Suppose that we delete a

customer. Now any query for any list of customers is potentially invalid, so we can say that all queries with the prefix `"customers"` should be invalidated in the cache; TanStack Query will automatically find those queries and only those queries. We should also invalidate the cache for fetching that particular customer but not the cache for fetching each single customer, which means that we'd still have all the data readily available.

We need to display the data from the server when it arrives, of course, but we might also want to display some kind of loading information. The response returned from the `useQuery` hook is an object with a lot of properties, but we're going to use only a few of them. The most important one is `data`, which is the data returned from the server. But remember that the server responds asynchronously, so the first time around, the data is probably `undefined`.

Suppose that we want to show all the customers, using a `Customer` component for each, but if they haven't loaded yet, we'll show the message `"Loading..."`. We can do that by extracting the `data` property as well as the `isLoading` property from the return value:

```
function Customers() {
  const { data, isLoading } = useQuery({     #1
    queryKey: ["customers"],  #1
    queryFn: getCustomers,    #1
  );    #1
  if (isLoading) return "Loading...";     #2
  return (
    <>
      {data.map((customer) => (    #3
          <Customer key={customer.id} {...customer} />
      ))}
    </>
  );
}
```

#1 We initiate a fetch for all customers and extract data and isLoading from the return object.

#2 If isLoading is true, the data hasn't arrived yet.

#3 We can use the data safely here because we know that it has been loaded from the server.

In a more concrete way, we would want to create a general hook for loading the list of things, `useAllThings()`.

Listing 9.7 A plain query

```
import { useQuery } from "@tanstack/react-query";
import { loadThings } from "./api/api";
export function useAllThings() {
  const { data = [] } = useQuery({
    queryKey: ["things"]
    queryFn:, loadThings,
  });
  return data.map(({ id }) => id);
}
```

That query is all it takes. We can use this `useAllThings` hook in many places, and TanStack Query will make sure to either serve data from the cache or load fresh data, depending on other factors (which I'll get to soon).

MUTATING

Mutations are surprisingly simple; we pass some data to a function and do "something" when the request succeeds. This time, we don't need to worry about query keys because no cache is automatically created or updated from mutations.

Mutations almost always take an argument but sometimes don't (in the case of a logout mutation, for example). Thus, we can pass in a function that takes an argument, such as for a delete-customer mutation:

```
const deleteMutation = useMutation({
  mutationFn: deleteCustomerById,
});
```

The return value from `useMutation` is an object that has a `.mutate` method, which is what we want to invoke. When the user clicks the delete button, for example, we need to invoke the delete mutation:

```
import { useMutation } from "@tanstack/react-query";
function DeleteCustomer({ id }) {
  const deleteMutation = useMutation({
    mutationFn: deleteCustomerById,
  });
  return <button onClick={() => deleteMutation.mutate(id)}>Delete</button>;
}
```

The second part of a mutation happens when the mutation has completed. You can pass in an `onSuccess` callback as an extra option to the `useMutation` hook that will be invoked when the request succeeds. A common thing to do is invalidate one or more queries, which are now considered invalid based on the mutation. If we delete a customer, we want to invalidate any query with the prefix `["customers"]` as well as any query for the customer with this specific ID.

To invalidate a query, we need access to the query client. We can use the aptly named `useQueryClient` hook, so this delete-customer button becomes

```
import {
  useQueryClient,
  useMutation,
} from "@tanstack/react-query";
function DeleteCustomer({ id }) {
  const queryClient = useQueryClient();
  const onSuccess = () => {
    queryClient.invalidateQueries({    #1
      queryKey: ["customers"],    #1
    });    #1
    queryClient.invalidateQueries({     #2
      queryKey: ["customer", { id }],   #2
    );    #2
  };
  const deleteMutation = useMutation({
    mutationFn: deleteCustomerById,
    onSuccess,
  });
  return <button onClick={() => deleteMutation.mutate(id)}>Delete</button>;
}
```

#1 We invalidate any query with the string "customers" as either the full key or a partial key.

#2 On the second line, we invalidate only queries with both the string key customer and the property id set to this particular customer.

For this concrete case, we want to create hooks for the various mutations required throughout the application, such as a hook for adding a new thing, `useAddThing`. The following listing shows the code.

Listing 9.8 A simple mutation

```
import {
  useQueryClient,
  useMutation,
} from "@tanstack/react-query";
import * as API from "./api";
export function useAddThing() {
  const queryClient = useQueryClient();
  const onSuccess = () =>
    queryClient.invalidateQueries({ queryKey: ["things"] });
  const { mutate: addThing } = useMutation({
    mutationFn: API.addThing,
    onSuccess,
  });
  return addThing;
}
```

The remaining hooks are basically slight variations. The `doThing` and `undoThing` hooks require the query to invalidate as an argument because they're used in both list view and detail view.

Listing 9.9 A slightly more complex mutation

```
import {
  useQueryClient,
  useMutation,
} from "@tanstack/react-query";
import * as API from "./api";
export function useDoThing(queryKey) {    #1
  const queryClient = useQueryClient();
  const onSuccess = () =>    #2
    queryClient.invalidateQueries({ queryKey });   #2
  const { mutate: doThing } = useMutation({
    mutationFn: API.doThing,
    onSuccess,
  });
  return doThing;
}
```

#1 The hook takes an argument, which is a query key.

#2 We pass the given query key to be invalidated.

With this listing, we have all the parts we need to migrate the application to TanStack Query, so let's get to the final implementation.

9.3.3 Implementation

Again, we'll study only the key files. The overall structure is similar to the preceding version, with basically the same files, albeit organized a bit differently.

One final bit of functionality is still missing. We need to maintain some UX state: what the current thing is, if any, as the server does not remember the current thing for us. That is, are we in list view or detail view, and if so, what is the `id` of that thing? We will use zustand for this task because it is the easiest library to use for such a tiny application.

FILE STRUCTURE

The new file structure looks like this:

```
src/
  backend/
    index.js     #1
  data/
    api/
      api.js     #2
      useAddThing.js     #3
      useDoThing.js   #3
      useLoginSignup.js    #3
      useLogout.js    #3
      useRemoveThing.js   #3
      useUndoThing.js   #3
    DataProvider.jsx    #4
    index.js    #1
    Loader.jsx     #5
    useAllThings.js   #6
    useCurrent.js    #7
    useHasCurrent.js #7
    useThatThing.js #8
    useThisThing.js  #8
    useUser.js   #6
  view/
    AddAThing.jsx   #9
    AllThings.jsx    #9
    Button.jsx    #9
    Form.jsx     #9
    Grid.jsx     #9
    index.js     #9
    Login.jsx    #9
    LoginSignup.jsx    #9
    Logout.jsx     #9
    Progress.jsx     #9
    RemoveButton.jsx     #9
    Signup.jsx     #9
    SingleThing.jsx     #9
    Thing.jsx     #9
    Things.jsx    #9
    ThingTitle.jsx     #9
    View.jsx    #9
  App.jsx     #9
  main.jsx  #9
```

#1 The backend and the data index file are both unchanged.

#2 api.js contains the underlying API calls using fetch(), and both are written in plain JavaScript.

#3 All the mutation hooks also live inside the API folder, but they're all similar to what you saw earlier.
#4 The data provider is now a query client provider required to make all the TanStack Query hooks work.
#5 The loader has been updated so that it returns null if no loading (no query, no mutation) takes place; otherwise, it renders the loader.
#6 These two hooks use a plain query.
#7 Because we need a single tiny stateful variable, we created a small hook using zustand.
#8 These two hooks are still the most complex bits, as they combine queries and mutations for use by their respective components.
#9 All these files are mostly identical to the preceding edition, with only minor changes where relevant.
#10 All these files are mostly identical to the preceding edition, with only minor changes where relevant.

SOURCE CODE

We saw how a query works in listing 9.7 and how a mutation works in listings 9.8 and 9.9. In this section, I'll only show the main API file, which contains all the underlying API fetch calls, as well as `useThatThing` and `useThisThing`, which are the two most complex hooks in the application.

First is `api.js`, shown in the following listing. It's structurally similar to the previous version except that it's pure JavaScript, with no stateful variables involved.

Listing 9.10 `src/data/api/api.js`

```
const URLS = {
  SESSION: "/api/session",
  USER: "/api/user",
  THINGS: "/api/things",
  THING: (id) => `/api/things/${id}`,
  DONES: (id) => `/api/things/${id}/done`,
  DONE: (id, did) => `/api/things/${id}/done/${did}`,
};
const wrappedFetch = async (...args) => {
  const res = await fetch(...args);
  if (!res.ok) {
    throw new Error("Unauthorized");
  }
  return res.json();
};
const get = (url) => wrappedFetch(url);
const post = (url, data) =>
  wrappedFetch(url, {
    method: "POST",
    body: data && JSON.stringify(data),
  });
const remove = (url) => wrappedFetch(url, { method: "DELETE" });
// USER API     #1
const getUser = () =>      #1
  get(URLS.USER).catch(() => false);      #1
const login = (data) =>     #1
  post(URLS.SESSION, data);      #1
const signup = (data) =>
  post(URLS.USER, data);    #1
const logout = () =>      #1
  remove(URLS.SESSION);     #1
// THING API     #1
const loadThings = () =>      #1
  get(URLS.THINGS);      #1
const loadThing = (id) =>      #1
  get(URLS.THING(id));     #1
const addThing = (data) =>      #1
  post(URLS.THINGS, data);      #1
const removeThing = (id) =>      #1
  remove(URLS.THING(id));
const doThing = (id) =>     #1
  post(URLS.DONES(id));      #1
const undoThing = ({ id, did }) =>      #1
  remove(URLS.DONE(id, did));    #1
```

```
  export {
    getUser,
    login,
    signup,
    logout,
    loadThings,
    loadThing,
    addThing,
    removeThing,
    doThing,
    undoThing,
  };
```

#1 In this file, we simply export the 10 underlying commands required to use the entire API.
#2 In this file, we simply export the 10 underlying commands required to use the entire API.

Next is the hook rendering a single thing in the list of all things, `useThisThing`.

Listing 9.11 `src/data/useThisThing.js`

```
import { useQuery } from "@tanstack/react-query";
import { loadThings } from "./api/api";
import { useDoThing } from "./api/useDoThing";
import { useUndoThing } from "./api/useUndoThing";
import { useCurrent } from "./useCurrent";
export function useThisThing(id) {
  const { data } = useQuery({    #1
    queryKey: ["things"],    #1
    queryFn: loadThings,    #1
  });    #1
  const thing = data.find((t) => t.id === id);    #1
  const doThing = useDoThing(["things"]);     #2
  const undoThing = useUndoThing(["things"]);   #2
  const seeThing =     #3
    useCurrent((state) => state.seeThing);   #3
  return {
    thing,
    doThing: () => doThing(id),
    undoLastThing: () => undoThing({ id, did: "last" }),
    seeThing: () => seeThing(id),
  };
}
```

#1 We query for the list of things, but because we know that we have this cached at this point, the library won't query the server for it. TanStack Query retrieves the latest response from the cache instead. Then we extract this particular thing based on the given id.
#2 The two hooks for getting the callbacks for doing and undoing things need the query key to invalidate postmutation, which is an array with the prefix "things".
#3 To switch to detail view for a single thing, we need an action function from the stateful UI hook, useCurrent, that we made with zustand.

Finally, we have `useThatThing`, which is the API hook that renders the detail view.

Listing 9.12 `src/data/useThatThing.js`

```
import { useQuery } from "@tanstack/react-query";
import { loadThing } from "./api/api";
import { useCurrent } from "./useCurrent";
import { useDoThing } from "./api/useDoThing";
import { useUndoThing } from "./api/useUndoThing";
import { useRemoveThing } from "./api/useRemoveThing";
export function useThatThing() {
  const id =     #1
    useCurrent((state) => state.currentId);    #1
  const seeAllThings =    #1
    useCurrent((state) => state.seeAllThings);    #1
  const doThing = useDoThing(["currentThing"]);     #2
  const undoThing = useUndoThing(["currentThing"]);    #2
  const removeThing =     #3
    useRemoveThing(seeAllThings);    #3
  const { data: thing } = useQuery({     #4
    queryKey: ["currentThing", { id }],   #4
    queryFn: () => loadThing(id),    #4
  });
  return {
    thing,
    doThing: () => doThing(id),
    undoThing: (did) => undoThing({ id, did }),
    undoLastThing: () => undoThing({ id, did: "last" }),
    removeThing: () => removeThing(id),
    seeAllThings,
  };
}
```

#1 First, we have to extract some data and functionality from the stateful UI hook made with zustand in useCurrent.
#2 Next, we include the two mutations to do and undo things. Here, we tell them to invalidate any query with the key prefix "currentThing".
#3 The mutation to remove a thing needs to know what to do next, which is update the stateful UI variable to trigger the application to go back to list view.
#4 Finally, we make the main query: fetching the information about the current thing with the given id.

You can see the full source code and test the application in the repository.

EXAMPLE: REACTIVE

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

```
$ npm run dev -w ch09/reactive
```

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/ch09-reactive**.

9.3.4 Bonus side effects

Besides built-in state, easy cache invalidation, more transparent flow, and other benefits that we've already described and used in this chapter, TanStack Query offers several benefits:

- *Network- or window-based automatic refetching*—The first benefit is automatic refetching based on network connection or window focus. TanStack Query automatically detects when your internet connection is cut, and whenever the connection is reestablished, it refetches all live queries. The library does the same for window focus. If you unfocus the browser window (by switching to another program or another tab), TanStack Query automatically refetches all live queries when the window regains focus. Although neither of these benefits would be hard to implement on your own, it's nice to get them for free.
- *Time-based cache invalidation*—The second benefit is cache invalidation based on time. TanStack Query can refetch any stale query that hasn't been updated from the server for a set period of time. Although this feature isn't enabled by default, you can easily enable it. You could set this interval to any value that makes sense for you. Five minutes would be a good start for this project.
- *Response deduping*—The third benefit, response deduping, is a bit more technical: if you have multiple components fetching the same query, TanStack Query pools them automatically, uses the same promise for all the instances, and returns the same data (referentially identical) to all the listeners.
- *Automatic request retries*—Finally, TanStack Query automatically retries any request that fails with a network or server error. This benefit

is particularly nice, as you might not think about adding such functionality until you need it, but TanStack Query has it built in.

9.4 Reactive caching with TanStack Query

This section introduces four principles that make working with remote data as a user smoother. All these principles reduce the delay from action to response—or merely the perceived delay:

- Updating the cache directly from a mutation
- Updating the cache optimistically based on expected results
- Preloading partial data where available
- Hiding the loader if partial or old data is available

I'll go over each of these principles in turn. At the end, I'll provide a link to the example for the goal-tracking application rewritten to follow these principles.

9.4.1 Updating cache from a mutation

If you make a mutation toward a single specific resource, and you know (by API specification) that it will update (or add or delete) only that specific resource, you could short-circuit the mutate-then-query flow and return the updated resource directly from the mutation. We can update the API to do that and use the response from the API in the mutation to update the cached query, which in turn will re-render any component using the original query. This process sounds complex but is simple, as you can see in figure 9.10.

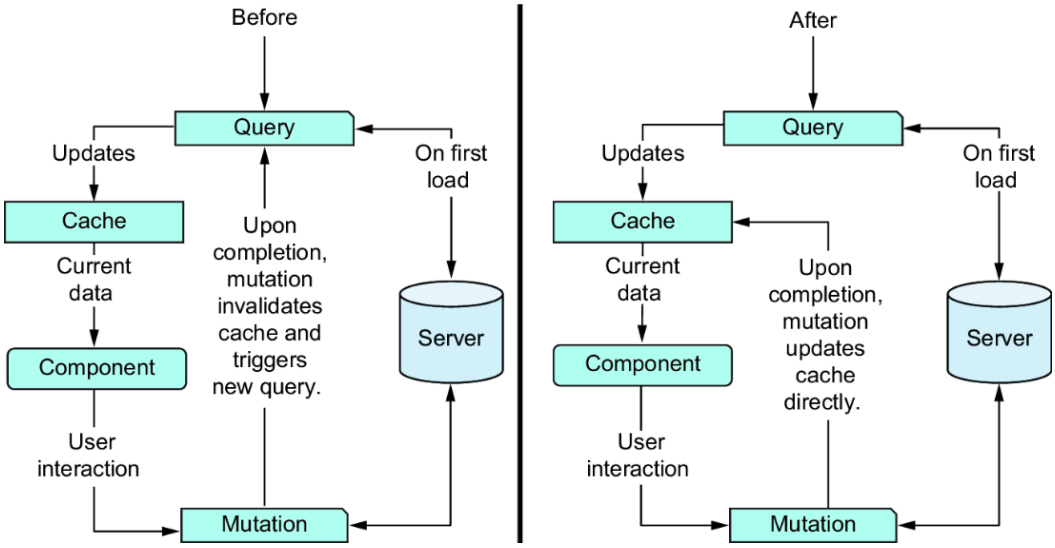

Figure 9.10 The flow when we update the cache query result directly from a mutation reduces server interactions by one for every mutation.

We can update the cache from a mutation in TanStack Query by passing an `onSuccess` callback as an option to `useMutation`. Let's see how this process works when we add a new thing in the `useDoThing` hook in listing 9.9. Note that we have to pass different callbacks in the two instances in which we use the variable because we have to update different queries (the whole list or the single current item), so the callback is a parameter. The following listing includes snippets from the various files to illustrate updating the cache from a mutation.

Listing 9.13 Updating cache directly from a mutation

```
// useDoThing.js (excerpt)
function useDoThing(onSuccess) {    #1
  const { mutate: doThing } = useMutation({   #1
    mutationFn: API.doThing,   #1
    onSuccess,   #1
  });    #1
  return doThing;
}
//
// useThatThing.js (excerpt)
const onSuccess = (newThing) =>      #2
  queryClient.setQueryData(    #2
    ["currentThing", { id }],   #2
    newThing,    #2
  );   #2
const doThing = useDoThing(onSuccess);
//
// useThisThing (excerpt)
const onSuccess = ({ name, done }) =>     #3
  queryClient.setQueryData(     #3
    ["things"],      #3
    (oldThings) =>    #3
      oldThings.map((oldThing) =>     #3
        oldThing.id !== id     #3
          ? oldThing     #3
          : { id, name, count: done.length }    #3
      )
    );
const doThing = useDoThing(onSuccess);
```

#1 We change the useDoThing hook to take an onSuccess callback as an argument so that we can vary what happens when we do a thing in list view compared with detail view.

#2 In detail view, we accept the response from the server (which is the newly updated thing) and replace the cache value. The useQuery in the same file automatically re-renders without querying the server. Note that we have to use the same query key as the original query, including the dynamic id value, to make sure that we update the correct cache value.

#3 In the other hook, useThisThing, we need to update the list query but update only the single item we're changing. Remember that in list view, we require the id, name, and count of a thing, so we make sure to include them in the new element.

This approach seems like a lot of work, but it saves us from making a full network request on every single increment, which saves a lot of round-

trip time in the long run. It not only makes the server less stressed but also makes UX a lot faster. Also, we can copy this logic to the `undoThing` hook as well, saving even more requests and making the experience even faster.

9.4.2 Updating the cache optimistically

A similar optimization is called *optimistic updates*. An optimistic update occurs when we update the cache prematurely as soon as we send the request with the expected result; then we refresh the data after the update as usual. If we did the optimistic update correctly, the user won't notice any difference between the optimistic update and the real update. Figure 9.11 illustrates this concept.

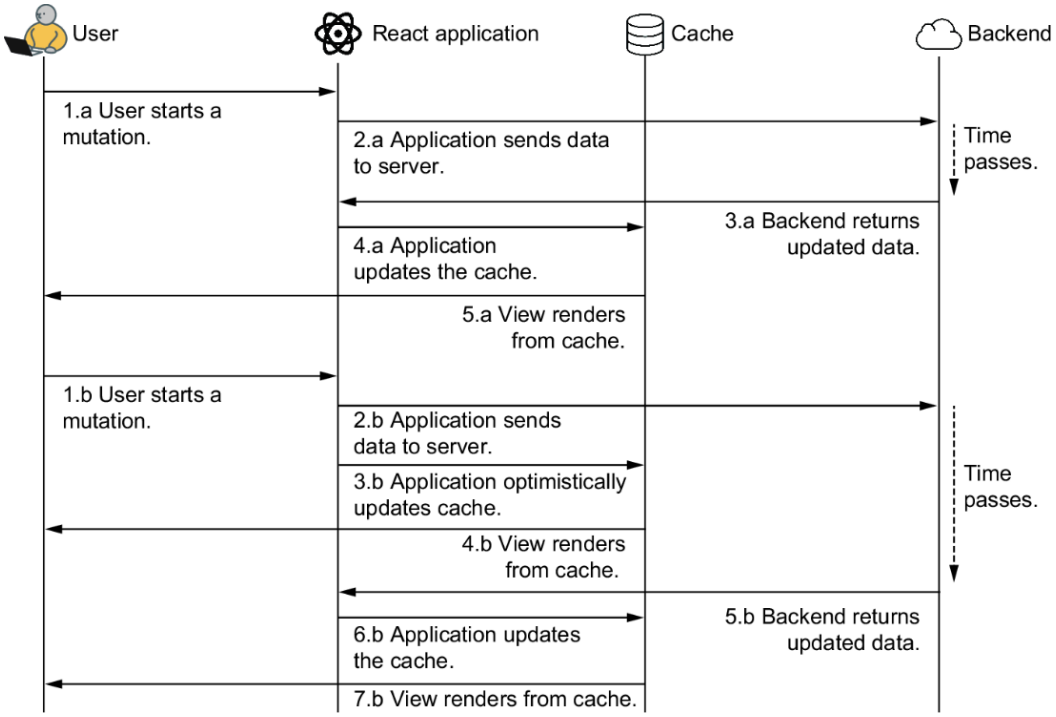

Figure 9.11 With optimistic updates, we update the local cache as soon as we send a mutation, so the application appears to update instantly. But we're ready to roll back the update if a server error occurs. In any case, we always refresh the state from the server after the update, so the optimistic cache value is purely cosmetic—never interacted with.

To accomplish optimistic updates, we have to do three things:

- When we send the mutation to the server, we also update the local cache with what we expect the server to do. If we add something, we can add the same thing locally; if we delete something, we delete the same thing locally, and so on.

- If the server request fails, we immediately roll back any optimistic updates that we performed and revert to whatever the cache state was before we started the request. A failed mutation is the same as status quo: nothing changes.
- If the request succeeds, we use the new data returned by the mutation to update the cache, replacing the optimistic update with the real server side–created data.

That last part—replacing the optimistic data with the real data—is often important. Our optimistic updates are naive, and we can make only the updates for which we have the data. On the server, more things might happen that we can't re-create in the client and wouldn't want to. We want to mimic the server action with as little effort as possible so that the user sees something in the 0.2 to 0.5 second it takes before the real server value comes in. That wait isn't long, so it's okay if the data is partial or not formatted correctly.

When we add a new thing in the goal-tracking application, for example, the server creates a new `thing` object with a random `id`. We can't guess what that `id` will be, and we don't need to. All we need to do is create a temporary item locally; that item will be close enough to the real thing, which will replace it in a very short time.

Achieving the preceding three steps is straightforward in TanStack Query. We pass a callback as an option for each of the three parts of the process:

- We can update the local cache as soon as the mutation is sent by passing an `onMutate` callback. This callback will receive the same data that the mutation function did. If we return something from the `onMutate` callback, it will be remembered internally as the *mutation context*, which will come in handy in the next step.
- We can roll back the optimistic update in an `onError` callback. This callback will receive not only the data passed to the mutation in the first place but also the mutation context returned by the `onMutate` function, if any. We use this context to remember what the cache looked like before we started messing with it, so we're able to roll it back.
- Most likely, the request will succeed, and then the `onSuccess` callback will be invoked. Here, we receive the newly created data from the server and can insert it into the cache instead of the temporary opti-

mistic data. If we did everything correctly, we won't notice the swap. The process is almost like magic.

Figure 9.12 illustrates this more concrete flow. Listing 9.14 shows how this process is implemented for the `useAddThing` hook.

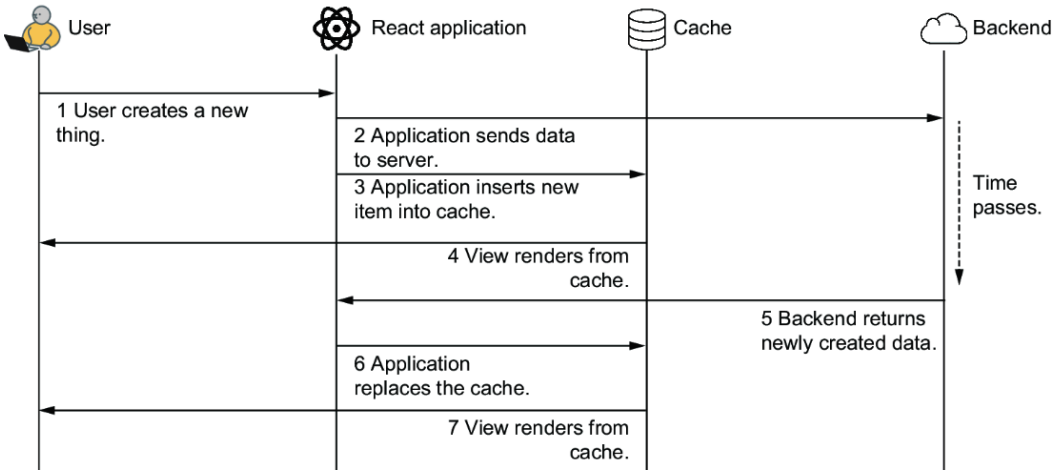

Figure 9.12 When the user adds a new thing to the list of things, we immediately and optimistically add the new thing to the local cache while the mutation is sent to the server. As soon as the real value is available from the server, the optimistic update is thrown out and replaced by the real data. Ideally, the user will not realize that the replacement took place.

Listing 9.14 Optimistic updates as soon as a mutation is sent

```
import {
  useQueryClient,
  useMutation,
} from "@tanstack/react-query";
import * as API from "./api";
export function useAddThing() {
  const queryClient = useQueryClient();
  const { mutate: addThing } = useMutation({
    mutationFn: API.addThing,
    onMutate: async (newData) => {     #1
      await queryClient.cancelQueries({ queryKey: ["things"] });
      const oldValue = queryClient.getQueryData(["things"]);
      const newThing =   #2
        { id: "temp", ...newData, count: 0 };    #2
      queryClient.setQueryData(["things"], (old) => [...old, newThing]);
      return { oldValue };    #3
    },
    onError: (error, data, { oldValue }) => {     #4
      queryClient.setQueryData(["things"], oldValue);   #4
    },   #4
    onSuccess: (data, variables, { oldValue }) => {    #5
      queryClient.setQueryData(["things"], [...oldValue, data]);
    },
  });
  return addThing;
}
```

#1 The onMutate callback is used to update the query cache with the new thing.
#2 Note that we initialize the count value manually to 0, as it is not part of the data we sent to the server, but we can predictably assume it to have a certain value when the real server response comes back.
#3 We return the old cache value from the onMutate callback to allow for a smooth rollback later.
#4 If an error occurs, the onError callback is invoked. Here, we can restore the original cache from the given context.
#5 Ideally, the request succeeds, and we can replace the optimistic value with the real one returned by the server.

For the fun of it, you can try to change the default count value used in the optimistic update of `newThing` in listing 9.14. Try swapping in this line:

```
const newThing = { id: "temp", ...newData, count: 20 };
```

You'll see that a new item is instantly added with the wrong count but is updated to the real value shortly thereafter.

9.4.3 Using partial data where available

When we list all the things, we get only some data about each thing because we need only some of the data that's available. When we want to display all the information about a single thing, we have to query the server for the full trove of data.

Yes, yes, I know—in this case, the difference is minimal. But in a larger application, there could easily be a significant difference in the amount of data loaded. Pretend that it would be too expensive to load all data in list view, okay?

If we're clever, however, we could use the partial data we already have to partially render the detail view while we wait for the full data set to come back from the server. In TanStack Query, this data is called initial data or placeholder data. The difference between initial data and placeholder data is that *initial data* is real data that could be used in place of the original, and *placeholder data* is dummy data that looks like the right thing but most likely isn't.

In other frameworks, this data is sometimes referred to as *skeleton data*. We could use practically anything as placeholder data, such as `Please wait, loading`, but because we know something about what we're loading, we can prefill the component with that information as initial data while we wait for the whole lot.

In this case, we need four pieces of information to display a single thing: the ID, the name, the description, and a list of timestamps of every time we did the thing. We already have the ID and the name. We also know how many times we did the thing, though we don't know when we did it each time. The description is missing.

So let's prefill the detail view component with the data we already have, trying to fill in as much as possible while we wait for the server to give us everything. We know that we'll have everything in about 0.2 second, so it doesn't matter if we're a bit off. If we can show something, the user will be happier than they'd be if they were staring at a blank loader. This process involves three steps:

1. Retrieve whatever partial information we have from the local query cache.
2. Augment the partial data as well as possible to simulate the full data set.
3. Use this augmented data as initial data for the query.

In this case, we want to show partial data when we go to detail view and load the thing information in the `useThatThing` hook. Figure 9.13 illustrates the process.

Listing 9.15 shows how this process is implemented in the `useThatThing` hook. Remember that before, we had this query:

```
const { data: thing } = useQuery(["currentThing", { id }], () => loadThing(id))
```

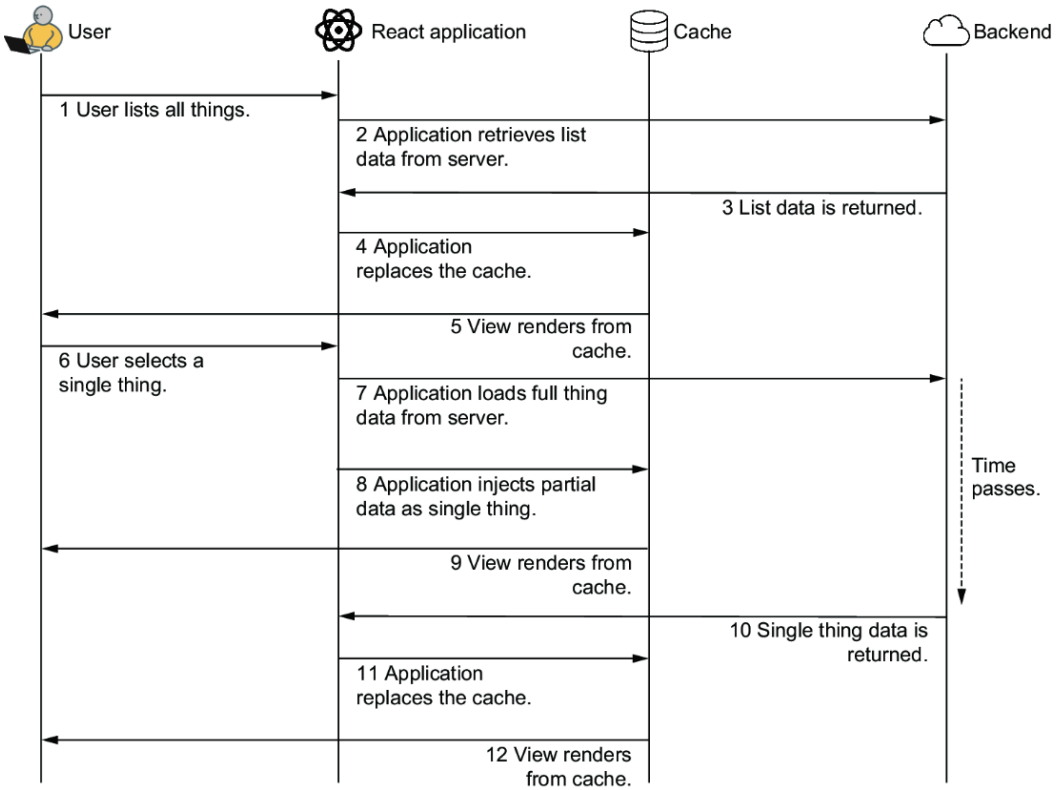

Figure 9.13 As we augment the partial data from the cache, we could put in whatever information we want for any piece of data we don't have. Often, empty data is desirable, but we could use some kind of loading indicator (dashes or dots).

Now, we pass in an extra option to the `useQuery` hook.

Listing 9.15 A query with placeholder data (excerpt)

```
function useThatThing(id) {
  ...
  // Find what we already know about this thing
  const queryClient = useQueryClient();
  const initialData = useMemo(() => {
    const things =
      queryClient.getQueryData(["things"]);     #1
    const { count, ...partialThing } =    #1
      things.find((t) => t.id === id);    #1
    return {    #1
      ...partialThing,    #1
      description: "...",    #2
      done: Array.from(Array(count))     #3
        .map((k, id) => ({ id })),    #3
    };
  }, [id, queryClient]);
  const { data: thing } = useQuery({

    queryKey: ["currentThing", { id }],
    queryFn: () => loadThing(id),
    initialData,    #4
  });
  ...
}
```

#1 We have to find whatever partial information we have for this item in the cache.
#2 We add a dummy description. We can't begin to predict what it will be, so something that indicates missing data is probably best.
#3 Next, we have to go from a count of entries to a list of "done" entries with an id each. We don't have a timestamp for each, so we don't prefill that data, and we make sure that grid view can handle missing timestamps.
#4 We pass this initial data object to the useQuery hook as an extra option.

The result is that we see a partial detail view for about 0.2 second as soon as we click the detail for a single thing, and when the server data arrives, we see the full result. Compare the two views in figure 9.14.

Figure 9.14 The partial data on the left is missing a description (instead showing some dots), and the timestamps are missing in grid view for all the thing's entries. Because this view is shown so briefly, however, it will suffice.

An added benefit is that we don't need to worry about what to do while we wait for the server data. TanStack Query automatically prefills the data value with the initial data and later re-renders the component with the data value set to the real server data when it arrives. Data is never `undefined` or missing. Our component is unaware of this process and renders whatever it receives.

9.4.4 Hiding the loader if some data is available

This last concept is purely cosmetic and doesn't involve anything technical. Remember that now we're doing a bunch of optimizations, and many requests are happening in the background while useful data is displayed onscreen. Sometimes, this data is only partial or optimistic, but it's still useful. For that reason, we can remove the loader in all but the few instances in which we have nothing to display—at this point, data is only completely missing just after login/signup, when we're requesting things from the server for the first time.

TanStack Query allows us to see the difference between what it defines as loading (when you need the data because you have nothing to show) and fetching (when you update the response of a query that you have only old or partial data for). We need to update the loader component, which we display whenever a query or mutation is ongoing. We can amend this component and pass a parameter to the `useIsFetching` hook, checking only for ongoing requests that are `"loading"` rather than merely `"fetching"`:

```
const isFetching = useIsFetching({
  predicate: (query) => query.status === "loading",
});
```

This change is a small one, but it makes a big difference in terms of perception. Not showing a loader makes an application feel faster even if we are still loading data in the background.

9.4.5 Putting it all together

I've implemented all four principles in the `better-reactive` example. Check it out to see how much faster it feels than the preceding version.

EXAMPLE: BETTER-REACTIVE

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

```
$ npm run dev -w ch09/better-reactive
```

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/ch09-better-reactive**.

I feel that this example provides much better UX, though I find the code somewhat cumbersome—especially the optimistic update combined with updating a single array entry in the `useAddThing` hook. The code is lengthy and not pretty, but it's fairly easy to read and improves the experience a great deal.

If you have a lot of optimistic updates in your codebase, you could create your own hook, `useOptimisticMutation`, on top of `useMutation`. This change would make the process smoother and the code easier to maintain.

Finally, I recommend switching to TypeScript for something like this example. The examples in this chapter are implemented in JavaScript to reduce the number of lines of code and details to explain. In the real world, it would be easier to work with in a typesafe environment such as TypeScript, which TanStack Query handles nicely.

Summary

- React applications most often communicate with a server to persist and read remote data.
- React is perfectly capable of dealing with remote data by itself, but as applications get larger, managing data cache gets more complicated.
- Several libraries make working with remote data easier, including TanStack Query. These libraries aren't network request libraries—merely data libraries built for working with remote data.
- TanStack Query takes managing local copies of remote data out of your hands and handles the work, including loading, refreshing, and invalidating.
- TanStack Query allows you to employ more complicated techniques to make remote-dependent applications easier and more pleasant to use:
  - Reactive caching enables your data layer to refresh data only when required and otherwise serve data from cache if it's known to be "fresh enough." Reactive caching is the core principle of many network data management libraries.
  - Another optimization technique is optimistic updates, which allow you to assume that a data mutation will work as intended on the server and temporarily update the cache locally (potentially only partially) while the mutation is sent to the server. When the mutation goes through and the server responds, we can update the cache to the real values resulting from the mutation, which might match what we expected or might be completely different.
  - We can use whatever data we have while we wait for the server to supply the rest. If you go to the details of a book in a list of books, you probably already have the title and author, so you can fill in that much data on the next page while you wait for the server to return the full response for the entire book data object.
  - We show a visual loader more selectively to make the application seem faster. Because we have something lifelike to show as soon as the request is sent, we can stop showing the loader and pretend that the request happened instantaneously. If we did everything correctly, UX is as good as with an instantaneous response.