11 React website frameworks

This chapter covers

- Rendering React on the server
- Developing a fullstack application in Next.js and Remix

React website frameworks allow us to run React on the server. You may wonder why you would want to, and both the short and the long answers are *speed* and *performance*. Your page renders a lot faster and your website becomes a lot more performant. These results are good for visitor retention, search engine optimization, and overall user experience—and most likely will also be good for your (or your employer's) bottom line.

So how does rendering React on the server make the application faster? It seems to involve simply moving the work from one computer (yours) to another (the server). Well, the reasons are many and fairly technical, so I'll dive into them in section 11.1. For now, let's summarize the reason: short-circuiting a lot of data transfer round-trips and even bypassing JavaScript to render the page to the visitor in as few milliseconds as possible.

The difference can be huge. Almost any website will be faster if it uses a React website framework (correctly!) rather than a client side–only React application, and some websites get orders of magnitude faster. It's not uncommon for a page to go from being ready in more than 5 seconds to being ready in less than 1 second. Now, that's good news!

In this chapter, I'll go over these technical details and discuss what makes server-side rendering (SSR) of React faster in general. Then I'll show how we can make a server side–rendered React application for a

weather app in two popular React website frameworks: *Next.js* (**https://nextjs.org**) and *Remix* (**https://remix.run**).
I could have dedicated this entire book to either framework, but as I'm going to cover both frameworks in a single chapter, I'll have to skip some details. Both frameworks have excellent documentation, though, so please follow the preceding links to learn more if you need more context.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch11**.

11.1 What's a website framework?

React is a JavaScript framework, and browsers run JavaScript, so browsers can run React. But environments other than browsers also run JavaScript, so other environments, such as Node on a web server, can run React.

The fact that you can run React anywhere that supports JavaScript is the cornerstone of a React website framework. You run React on the server, so you can generate HTML on the server by using React as a templating language. Running React on the server gives you several benefits, including the following:

- *Fullstack development*—This type of development enables you to build your entire website in a single application with frontend and backend merging. You don't have to have two different projects or codebases or even two different teams that need to coordinate. Everything is built together, so you have optimal colocation. You can directly change what is loaded from the database and where it goes in the final HTML in a single file, even though those things happen in the backend and frontend, respectively.
- *SSR*—With SSR, HTML is served instantly from the server, improving page-load times, which might in turn improve both visitor retention and search engine ranking.
- *Dynamic content*—You can integrate any content into your website seamlessly because you control the whole stack. You can load content from a database or from an external API, even with some se-

crets that you normally wouldn't put in your frontend application. The framework will make sure that your content is passed correctly from backend to frontend; you don't have to do anything.
- *URL routing*—URL routing allows you to use the URL as a source of navigation. Complex routing rules are built into most frameworks, making it easy to create pages.

But the benefits come with one particular requirement: hydration. When the client renders the React output for the first time, every single byte of output has to match that of the server rendering.

The rest of this section touches on these concepts in general. When we understand what makes a React website framework tick, we'll discuss the two candidate frameworks in section 11.2.

11.1.1 Fullstack React as a concept

Normally, React is a frontend-only technology, limited by the normal restrictions by which frontends abide. These restrictions include the following:

- You need a separate backend developed next to or even separate from the front- end to be able to talk to a database or any similar form of shared storage.
- Everything in the codebase is public knowledge, so you cannot use secret API keys or similar content that you don't want strangers to access (because they might be able to extract private data from your application).
- You need an agreed-upon API to communicate between the React application and the backend. If you update some view in the front-end, you may need new data from the backend, so you have to update the API and update the backend to accommodate this change.
- Deep linking (with HTML rendered on the server) is next to impossible.

When you introduce fullstack React, however, you eliminate all of these problems because you run React on both the frontend *and* the backend. These problems suddenly vanish:

- You develop the frontend and the backend as one single application, so you don't have to spend much time thinking about accessing a database or saving files to permanent storage.
- You can easily annotate which parts of the code are compartmentalized to run only on the client or only on the server, so you can hide your API keys on the server with little effort.
- API worries go away because you literally write the frontend using the data and the backend providing the data in the same file.
- Deep linking is trivial and built into the platform.

Compare what happens underneath the UI when you visit the page for a specific movie on a movie-database website in two different setups. In setup A, we run React as frontend only, and we have a separate backend written in another language (which could .NET, Python, or COBOL for all we care). In setup B, we run React as part of a fullstack React framework such as Next.js or Remix. We are going to visit a detail page somewhere deeper on the website (so, not the front page). Figure 11.1 illustrates what happens in each setup.

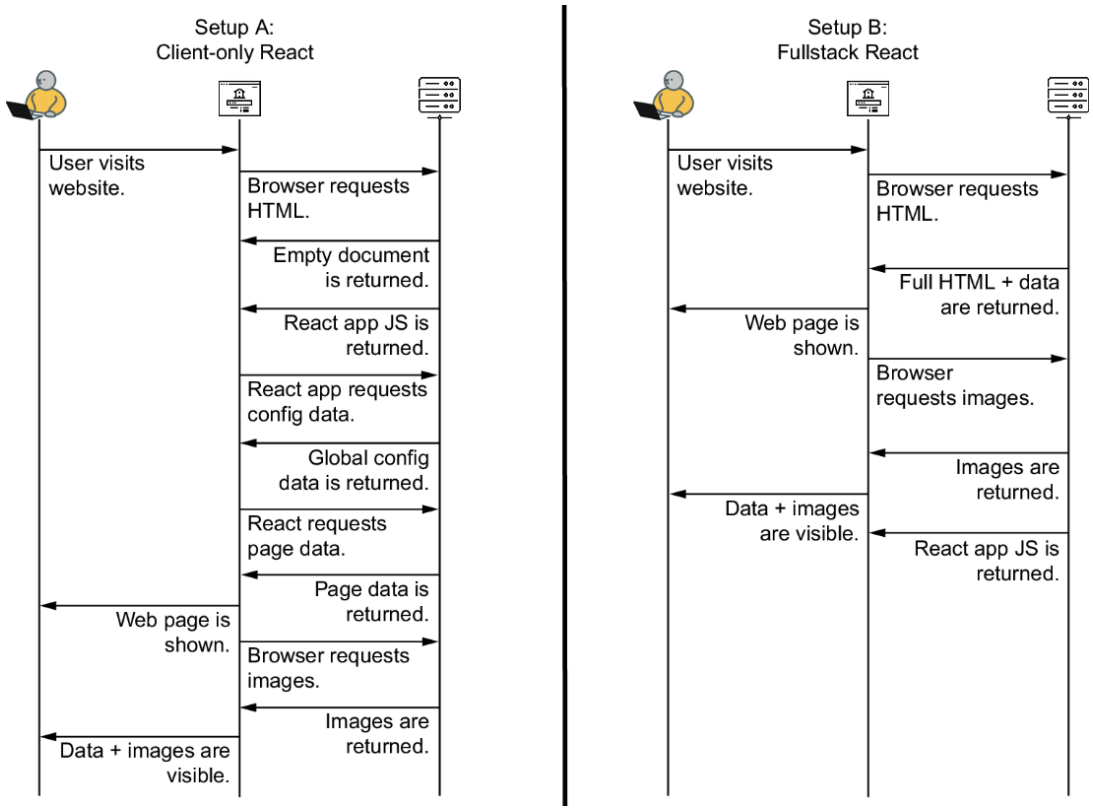


Figure 11.1 In setup A, which has a classic backend, we need a lot of communication and setup between the two parts of the application before the page can render. In setup B, we return everything in one go, and the client is ready to serve the proper content instantly.

One of the coolest things about fullstack React is that, because the HTML comes prerendered and fully ready to display the entire page from the server, JavaScript isn't required to display that page. As you see in the last step of setup B in figure 11.1, the JavaScript for the React application can be loaded after the page is shown to the user. Yes, the React application is required to make the page fully interactive, but that behavior isn't required for the first display, making the website even faster.

11.1.2 Rendering HTML on the server

We've been using Vite for all our applications so far, and it's great for playing around with React, but it's bad at doing one thing: serving relevant HTML from the server. You may remember that we have a file called `index.html` in all our Vite-based applications. This file is the same one (with some scripts and stylesheets inserted) that will be served from the server and loaded in the browser. You may also remember that this file is empty for all intents and purposes. When we ignore nonvisual elements (such as scripts and meta tags), the file is

```
<!doctype html>
<html lang="en">
  <body>
    <div id="root"></div>
  </body>
</html>
```

That's it. The file contains an empty `<div>`. If you use an old phone, have JavaScript disabled, are on a slow connection, or if the scripts fail to load for some reason, you will see a blank web page.

With SSR, we render the full HTML on the server, which means that we run the entire React application with all the correct data put in all the correct components, take a snapshot of the resulting HTML, and return that HTML to the browser. Compare the new approach with the classic one in figure 11.2.

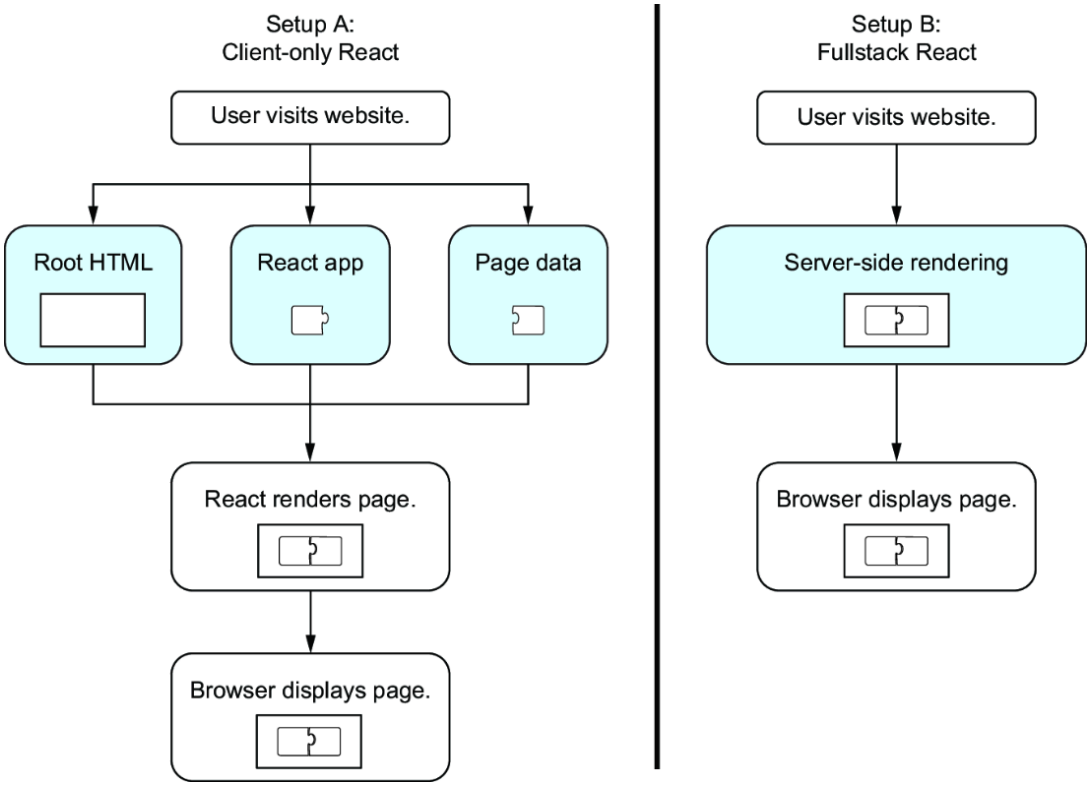
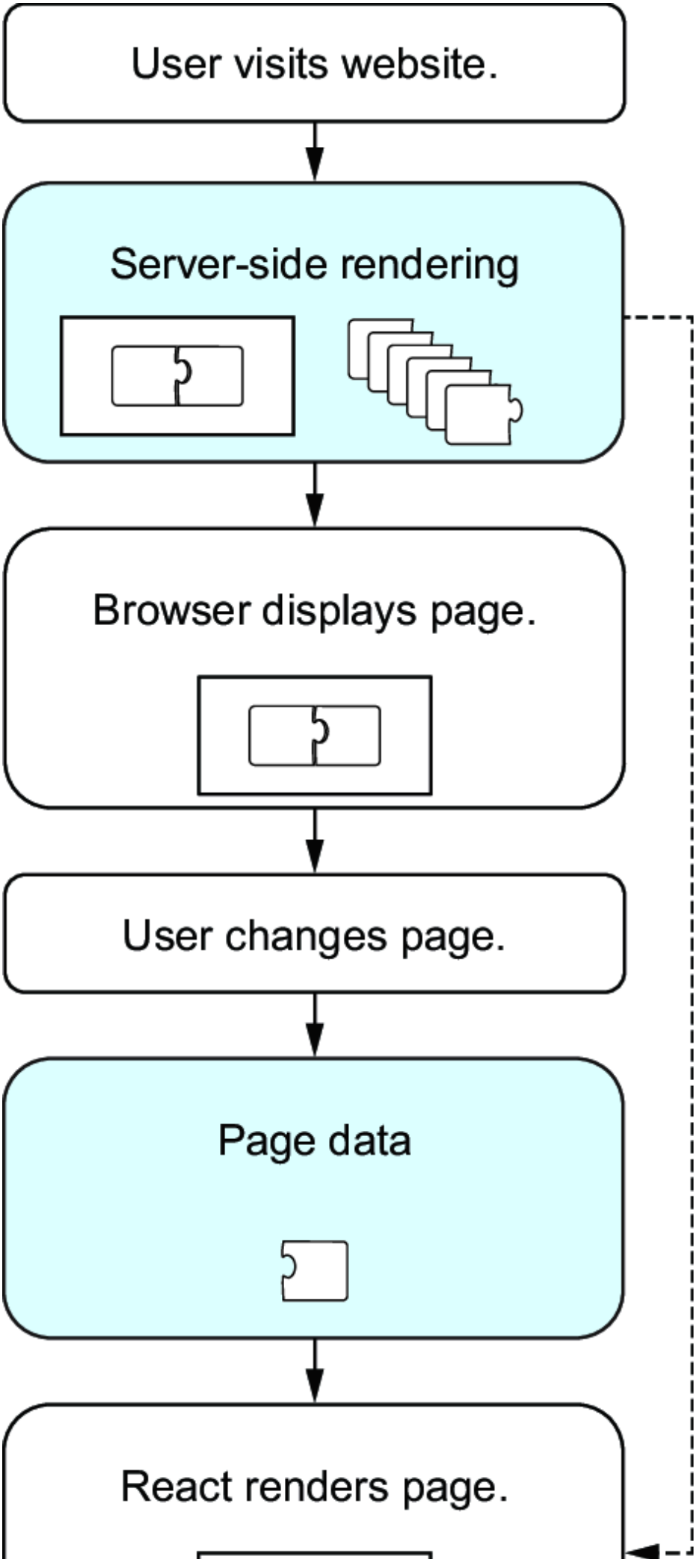


Figure 11.2 Imagine a page to display as the simplest puzzle ever. It has only two pieces: a component defined in React that defines the template and a data package that contains the data required to turn the template into HTML. In a classic React setup, we retrieve those two pieces in two separate requests and put them together in the browser when everything is loaded. In the framework setup, we get both pieces at the same time instantly.

One question about figure 11.2 may arise: what happens if we start on one page and then move to a different page that requires data from the server—do we have to wait for the server to render the full HTML for that page as well?

The answer, luckily, is no. The server render takes place for the first page. After that, only the data is loaded from the server. That's why, when we write components for SSR, we write them as two separate items: a component and a data package. For the first page in a visit, the two are combined on the server and returned along with the JavaScript application. But for subsequent page visits, only the data package is requested for each page, as we already have the React application (including the component) available in the browser, so all we need is the data for each page. Figure 11.3 illustrates a visit to a website.


User visits website.
Server-side rendering
Browser displays page.
User changes page.
Page data
React renders page.

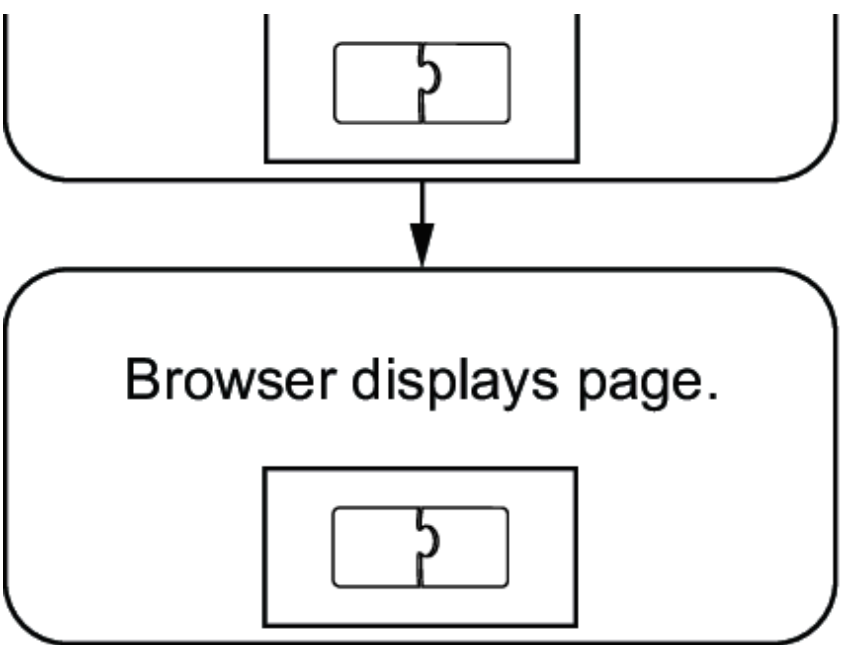


Figure 11.3 On the first page during a visit, the full HTML is rendered on the server, but for subsequent page visits, only the data package is returned. In puzzle lingo, we get the full puzzle for the first page on the first render and the template piece for every other page puzzle instantly. Then we have to retrieve only the data piece for each new page we have to render.

This concept of every page having two pieces—data and a React component—is central to the way React website frameworks function. That data piece can contain all sorts of dynamic content, as we will discuss next.

11.1.3 Dynamic content

Because we generate the data package for every page on the server, we can load whatever we want—information from a database, a file, or a third-party API—or generate new content. We can even combine any number of sources of data and collect the data into a single package that we use in rendering the page. The framework takes care of using that package on the server or on the client depending on how the page will render (as in section 11.1.2), so all we have to worry about is gathering that data.

We're not going to spend much time discussing various Node.js frameworks for working with data, but we will discuss a single library—a very popular database wrapper library named Prisma. Prisma is an object-relational mapping (ORM) library. Essentially, such a library is an abstraction library built on top of SQL (or even NoSQL), abstracting

away the complexities of writing SQL queries and instead using simple object notation for writing data to and retrieving data from a database. Remix comes with Prisma as the strongly recommended database engine, already installed in many of the example applications available for Remix. Next.js is a bit more open minded, having example applications that use other frameworks as well, but Prisma is still a strong recommendation, so we're going to use Prisma in this chapter and in chapter 12 too.

Prisma thrives in TypeScript because it elegantly makes your code type-safe even for complex queries, with the types being extracted from your database schema automatically. In this chapter, we'll use JavaScript instead of TypeScript, but for the project in chapter 12, we'll dive into typed database queries.

TIP Prisma is fairly complex but also well documented, so please check the documentation on their website (**https://www.prisma.io**) to learn more.

11.1.4 Hydration is necessary

I have to confess that one very important aspect is missing from figure 11.3. Let's amend that situation in figure 11.4 before we discuss why it's so important.

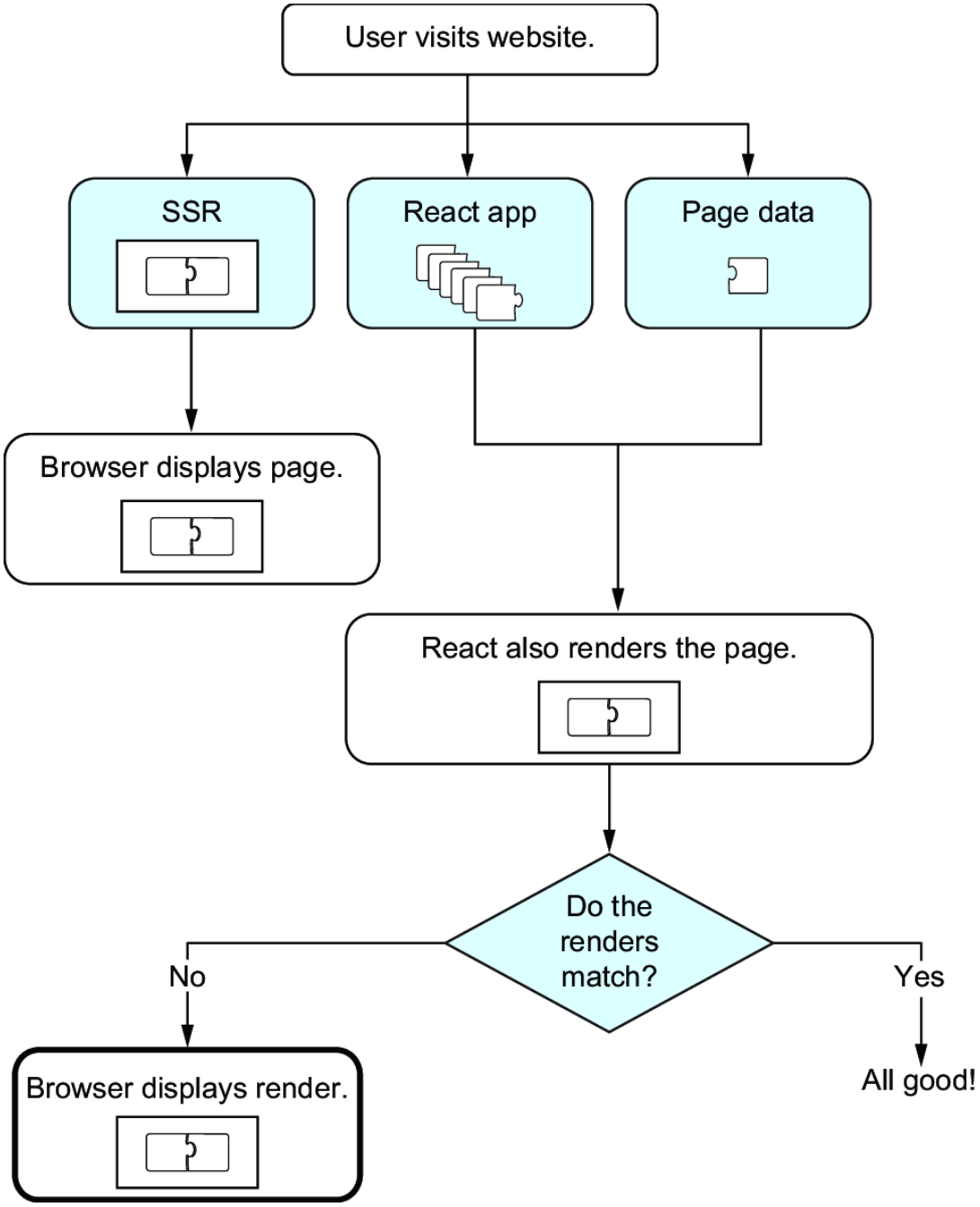


Figure 11.4 React also renders the HTML page and compares the React-rendered version with the server-rendered version. This process is called *hydration*, and only when the two HTML versions are identical is React happy. If they aren't, React forces the browser to render the React-created HTML instead, displayed in the box with the thick border at bottom left.

The step of figure 11.4 in which React also renders the page and compares the rendered HTML with the server-generated HTML, is called *hydration*. This step is important for gaining the performance boost that SSR offers. We'll discuss these implications in the next few sections.

ONLY PERFECT HYDRATION IS ALLOWED

For hydration to work correctly, you must have the same output down to the last byte. Nothing can differ; otherwise, React will complain. This situation results in several consequences, the most obvious one being that you cannot have anything random generated inside your application because it would affect the output.

If you want to display a random quote in a hero banner (or random ads on your blog), for example, you cannot make that random choice in React. This random choice has to happen on the server outside React and be sent into React in a deterministic fashion. You need to include the random choice in the data package and not let React choose (figure 11.5).

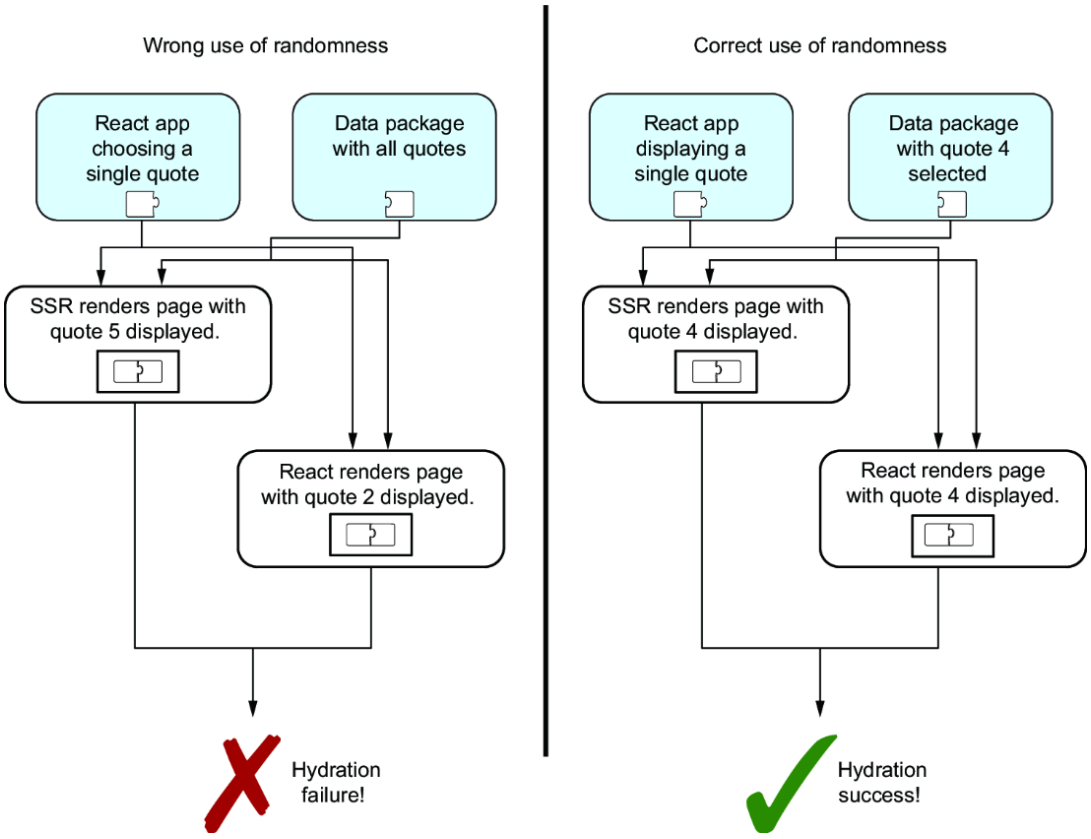


Figure 11.5 To display a random quote, we need to place the random source of information in the data package and then use that data on both the server and the client to ensure proper hydration.

WHAT'S THE PROBLEM WITH NONPERFECT HYDRATION?

You may wonder why hydration has to be pixel perfect—why it can't work if some parts differ slightly. That's just how React is built; it's an

all-or-nothing concept.

What happens if you do break this principle? Will the application break? No, nothing of the sort happens. Hydration will work, but with a performance penalty. Let me explain why that penalty occurs.

If hydration succeeds, React starts up faster than normal because it doesn't have to render to the document object model (DOM)—only check for any differences (which is faster than rendering). But if hydration fails, React clears the document and re-renders the whole page from an empty page, but only after checking for differences, which makes it slower—not by a lot, but still slower. Another factor is HTML file size. In a classic setup, the HTML file is tiny because it's empty. But when it's server rendered, the HTML file can be quite big if the page is complex. So only if hydration works correctly does server-generated React gain a performance boost. Compare the differences in figure 11.6.

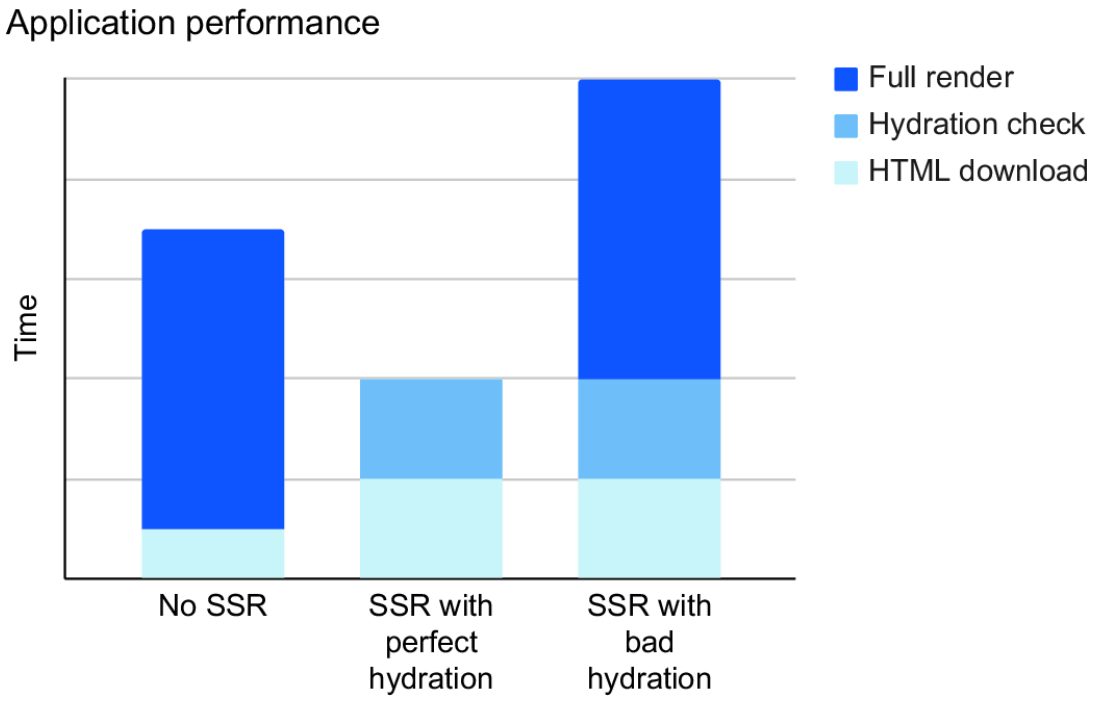


Figure 11.6 The performance boost of SSR happens only when hydration works perfectly. Otherwise, a significant penalty occurs. Note that this figure isn't an actual graph in milliseconds, only an illustration of the concept.

WHAT ABOUT PARTIAL HYDRATION?

Partial hydration is fairly new; it involves splitting content into smaller bits that can be hydrated or not individually. This topic is fairly com-

plex and well beyond the scope of this book, but I want to mention it in case you want to dive deeper.
A lot of modern React development happens in this arena, but it's quite complicated and requires more introduction that I can provide in this chapter. Topics such as React Server Components, streaming content delivery, server actions, tainted objects, and so on are part of bleeding-edge React frameworks, and most of these new features boil down to the benefits of partial hydration.

NOTE Several of the new features in React 19 are also related to server actions, and we will see big improvements in the performance of React website frameworks with the adoption of React 19. I won't be discussing these new features in this book but stick to full hydration only.

11.2 Implementations

With our newfound understanding of the challenges and benefits of React website frameworks in general, let's take a closer look at our two contestants to see what makes them special, how to get started, and how to get your hands on my implementations.

11.2.1 Next.js

First, we have Next.js, the *grand old man* in the React website framework category. Besides being a great product, with all the bells and whistles you want from a React website framework, Next.js is popular because of its primary sponsor. It's created and supported by Vercel, an online hosting service for (among other things) fullstack React websites. So their incentive is obvious: Next.js is optimized for hosting, speed, and performance all around. If you want to read more about the framework and its benefits, please check **https://nextjs.org**.

Next.js recently came out with support for partial hydration and React Server Components, but it still supports the old model, which we'll use exclusively in this book. In the Next.js documentation, these two approaches are known as App Router using React Server Components and Pages Router using only regular components. We'll stick to Pages Router, so make sure that when you're using the Next.js documenta-

tion, you're on the Pages Router section of the website: https://nextjs.org/docs/pages. See figure 11.7 for additional hints.

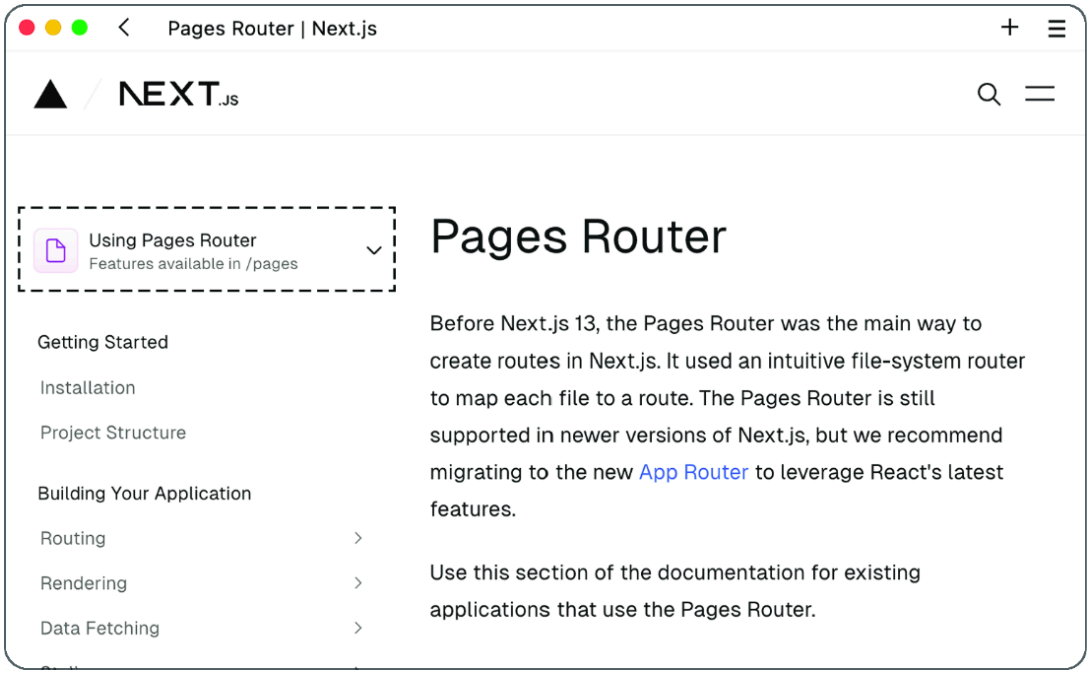


Figure 11.7 The Pages Router part of the documentation. If the menu heading is Using App Router, you can't use whatever information you read in a Pages Router–based application.

A CLEAN NEXT.JS PROJECT

To start a new, blank Next.js project, initialize it with the following command-line script, which is similar to how we create projects with Vite:

```
$ npx create-next-app@latest
```

This script asks you a couple of questions, including whether you want to use TypeScript. For this project, we're not going to use TypeScript, but I recommend it in general; Next.js is well typed, and using TypeScript is a great experience. The script will also ask whether you want to use the new App Router or the classic Pages Router. Choose Pages Router, which is the version we're using in this chapter. The result is a clean, empty Next.js application with a single default route (figure 11.8).

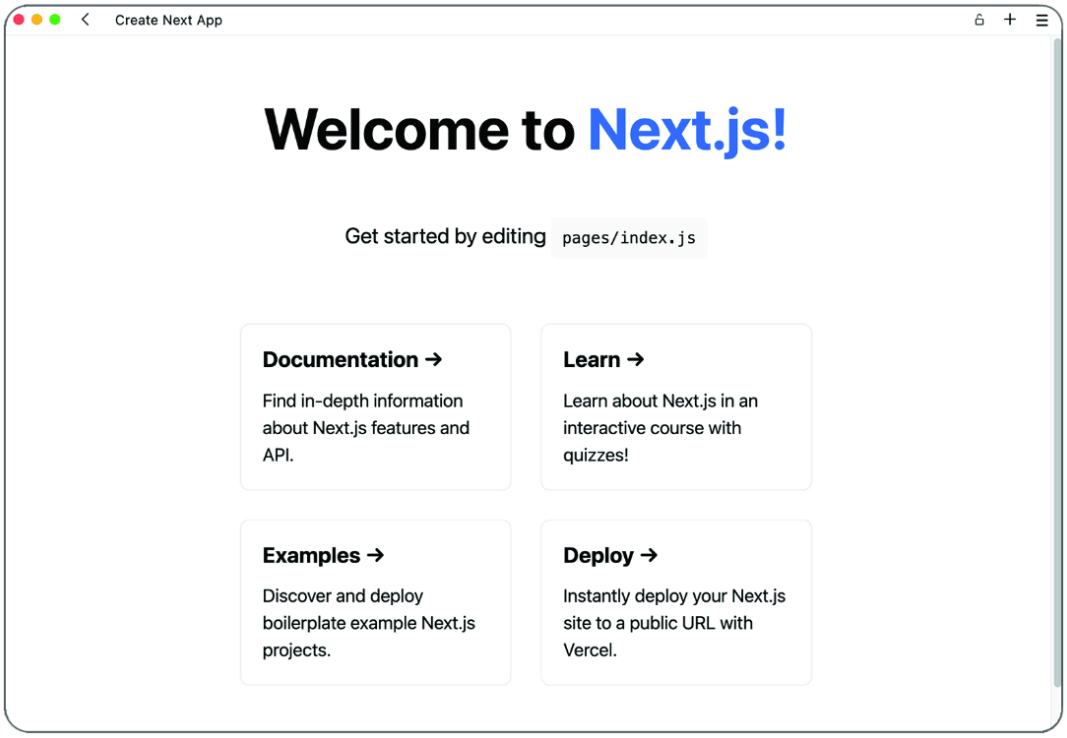


Figure 11.8 The default Next.js application comes with a single route and some helpful links.

MY SOLUTION

Rather than start with a clean slate, you may want to look at my example implementation of the weather app that we're building in this chapter. You can get it by checking out the `ch11/nextjs` example.

EXAMPLE: NEXTJS

This example is in the `ch11/nextjs` folder. Note that this example is nonstandard and doesn't use the regular Vite setup; it's a custom Next.js setup. Before you run this example locally, make sure to run the setup first (only once for this example):

```
$ npm run setup -w ch11/nextjs
```

Then you can run the example by running this command:

```
$ npm run dev -w ch11/nextjs
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch11-nextjs**.

11.2.2 Remix

Remix, also known as Remix.run, is a rather new tool for building React websites, but it has taken the community by storm and is quickly gaining traction. Remix was created by Ryan Florence, known as the creator of React Router, a popular routing library that sits at the core of Remix. Remix has since been acquired by Shopify, which chose to fully open source the project, so it comes with a lot of professional support and experience.

One of the main features of this library is its surprising underreliance on JavaScript. Yes, a correctly produced Remix web application will work even without enabling JavaScript in the browser. This JavaScript independence, of course, doesn't extend to interactive client side–only features such as dialog boxes and menus, but basic form handling and navigation work perfectly fine without JavaScript. In that sense, building Remix websites feels akin to building websites back in the old days before all the modern features came along, but using React as the wonderfully powerful templating engine.

There's a lot more to this framework, but the concept of making websites that work even without JavaScript enabled permeates the architecture and influences many of its features, which we'll see a lot more of later.

A CLEAN REMIX PROJECT

To start a new Remix project, as you do in Vite and Next.js, you call a command-line script:

```
$ npx create-remix@latest
```

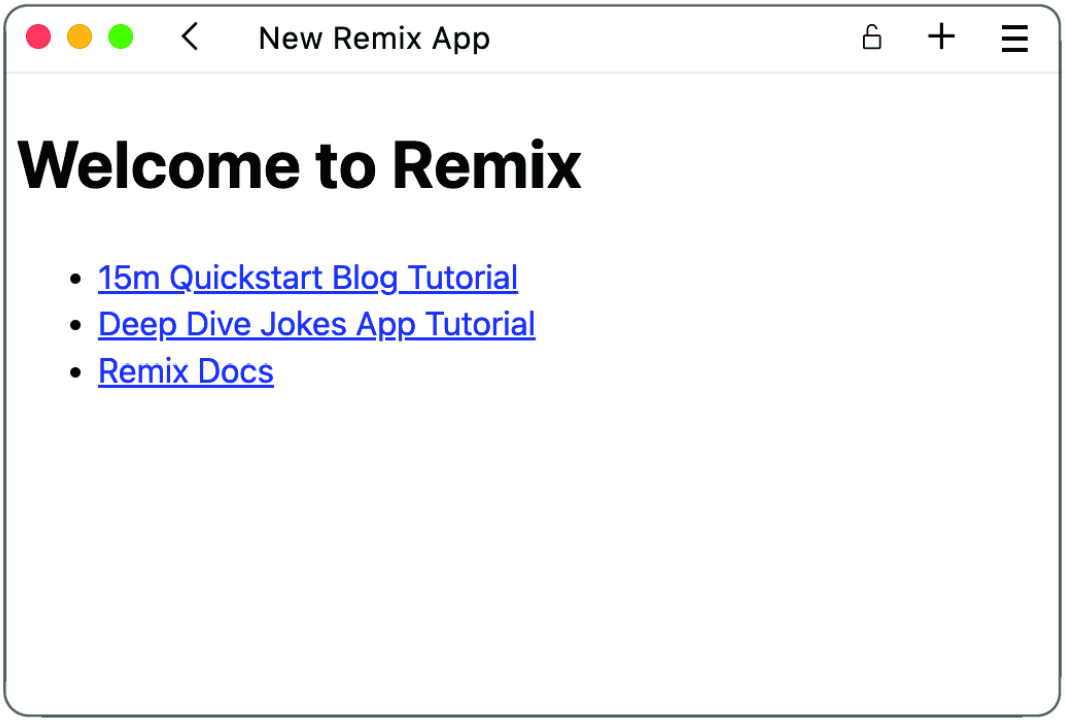


Figure 11.9 The default Remix application is a lot more bare-bones than the Next.js one but still contains a few useful links.

This script asks you some questions about the install, and then you're good to go. When the project is up and running, you'll see something like figure 11.9.

Remix comes with some starter templates called *stacks*. Stacks are collections of tools and principles that are commonly used in React development projects, so these tools can be all the things I've introduced in the past many chapters: styling frameworks, testing tools, caching libraries, data handling, and so on. You can check out the default stacks in the Remix documentation at **https://mng.bz/pp50**.

MY SOLUTION

Rather than start with a blank slate or a premade stack, you probably want to see my application, which implements the chapter 11 weather app. It's available in the `ch11/remix` example.

EXAMPLE: REMIX

This example is in the `ch11/remix` folder. Note that this example is nonstandard and doesn't use the regular Vite setup, but a custom Remix setup. Before you run this example locally, make sure to run the setup (only once for this example):

```
$ npm run setup -w ch11/remix
```

Then you can run the example by running this command:

```
$ npm run dev -w ch11/remix
```

Alternatively, you can go to this website to browse the code or download the source code as a zip file: **https://reactlikea.pro/ch11-remix**.

11.2.3 Environment values and API keys

Before I dive into the details of the app, you may notice that when you check out one of the preceding two examples, they don't work straight away. The folders are missing a few environment variables. These variables are supposed to be defined in a file named `.env` (for Remix) or `.env.local` (for Next.js), but this file is not included per convention because it includes local secrets you shouldn't share with others. Instead, I've provided an example file named `.env.example`. You should copy this file to `.env(.local)` and edit it to your liking.

The only value you need to change in this file is `OPENWEATHER_API_KEY`. The value is set to `"<YOUR_API_KEY>"` in the example file, which doesn't work. You need to go to the OpenWeather website, get your own API key, and insert it into your own `.env(.local)` file. Follow these steps:

1. Go to **https://home.openweathermap.org/users/sign_up** to sign up.
2. Click the confirmation link that you receive via email.
3. After confirmation, go to this page to see your API key: **https://home.openweathermap.org/api_keys**.

4. Copy the 32-character alphanumeric string and insert it between quotes in your `.env(.local)` file like so:

```
OPENWEATHER_API_KEY="ABC123...ABC123"
```

You need to change the other value, `DATABASE_URL`, in the `.env(.local)` file only if you want to use a different database from a simple SQLite file–based one. Otherwise, ignore the database setting. The Remix example also has a secret key used for cookie encryption. This key is not important to edit in this example, but you should edit it if you want to store private information in cookies.

The setup scripts for both examples also copy the example environment file for you, but you'll still have to enter a real key acquired from the OpenWeather website.

11.3 Let's create a weather app!

In this section, we'll implement a weather app that can load the weather for any location in the world. The final result will look like figure 11.10.

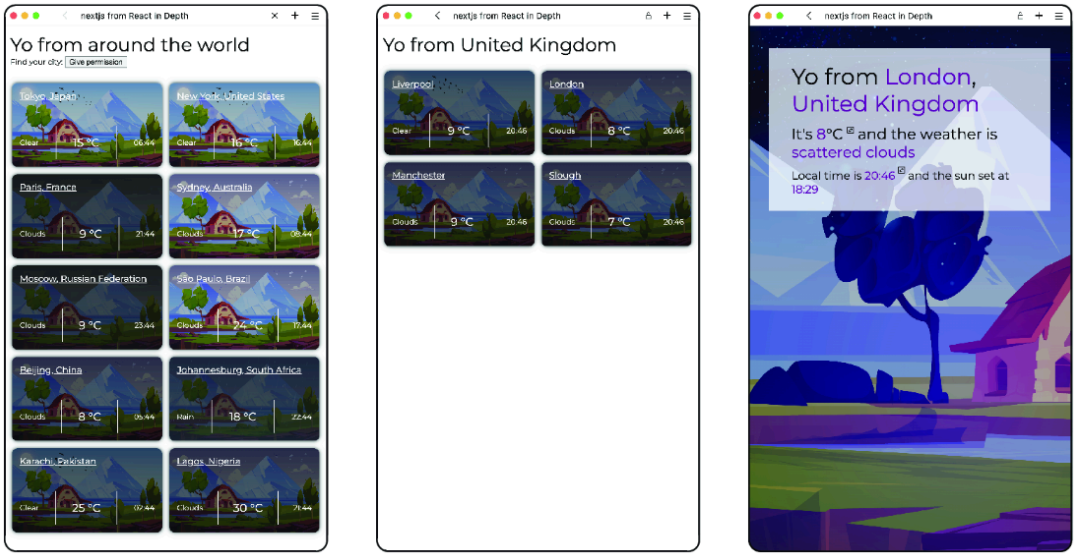


Figure 11.10 The final product, our weather application, as it will be implemented with Next.js and Remix. Notice that the application displays weather conditions around the world and has a front page, a country page, and a city page.

In this section, we'll go through the key parts of this application and discuss why this particular website will be a challenge. Then we're going to solve this challenge in both Next.js and Remix to compare and evaluate. These key parts are

- Using the URL to display different pages
- Loading data on the server and reusing that data on the client
- Persisting local state data in a cookie
- Creating an API that allows React to interact directly with the server

11.3.1 Using the URL

One crucial thing that the screenshots in figure 11.10 don't show is the URL. I've been taking screenshots in the extremely minimalist browser named Min, which hides the URL. This approach has been perfect so far, as the URL has always been `http://localhost:3000/`. For every app, every state, every view so far, we've always been at this URL. In this application, however, we need to see the URL as we navigate around. So let's repeat the screenshots in a slightly less minimalist browser, Brave (figure 11.11).

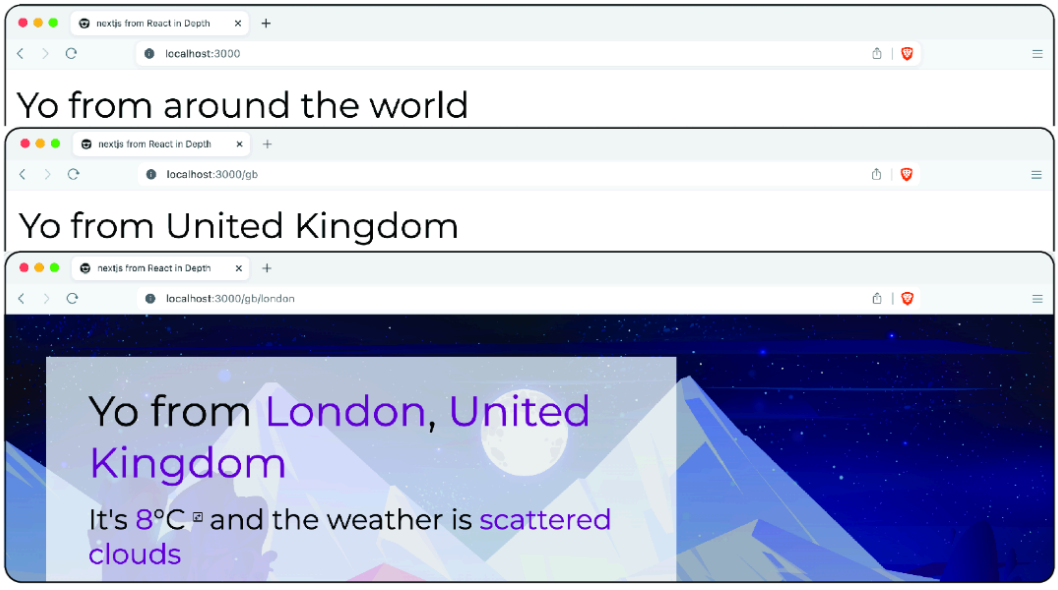


Figure 11.11 The same three views as in figure 11.10 with the URL visible. The root (`/`) is the front page, a country code (such as `/gb`) leads to the country page, and if we append the city name (such as `/gb/london`), we get the city page.

Easy URL mapping is one of the core features of most React website frameworks. Although normal React applications using Vite or Create

React App are built solely with components, most React website frameworks use both components and pages as their building blocks.
Pages in this context are still React components, but they're organized slightly differently. Their names are related directly to the URL, and they can be extended with some additional functionality.

Suppose that you want to have a page on your website named `/about`, where people can read about your website. In both Next.js and Remix (and in most other frameworks), this task requires simply placing a file named `about.jsx` in the `routes/` or `app/pages/` folder. (The folder structure varies a bit with the frameworks.) Then this file simply has to export a React component, which becomes the contents of the about page. If you want to put the page at `/about/team`, you create a component in a file named `team.jsx` inside the `about/` folder inside the `routes/` folder (in the case of Remix) or the `app/pages/` folder (in the case of Next.js).

Remix v2 came out in September 2023 and included a new flat routing scheme. Whereas Remix v1 used the same folder structure in the `routes` folder that was used in the URL, Remix v2 uses a flat structure with periods (`.`) as folder separators. The Remix v2 page component for the route `/about/team` becomes `/app/routes/about.team.jsx`.
One curious detail in Remix v2 routing is that the root index page is called `_index.jsx` rather than `index.jsx`. Table 11.1 summarizes the route system.

Table 11.1 Basic routes are resolved to straightforward files with file and folder naming.

	Component location		
URL	**Next.js**	**Remix v1**	**Remix v2**
`http://localhost:3000`	`/pages`	`/app/routes`	`/app/routes`
`/`	`/index.jsx`	`/index.jsx`	`/_index.jsx`
`/about`	`/about.jsx`	`/about.jsx`	`/about.jsx`
`/about/team`	`/about - /team.jsx`	`/about - /team.jsx`	`/about.team.jsx`

ROUTING DYNAMICALLY

But wait—do we have to create a folder for every country in the world, copy the same component into every one of these folders, and then create a file for every city in the world? That task would be impossible, so, no, we're not going to do that.

Instead, we're going to use dynamic routing, which both of our frameworks handle well though differently. The idea is to use parameterized URLs. You indicate, through naming, that a given file or folder should match any string of letters, and you can resolve it later.

Suppose that you have a team-member page for every person in your company, which has more than 200 employees, and you have all those employees listed in a database. You want this URL structure:

- `/about/team/jack-johnson`
- `/about/team/john-jackson`

But you want all those pages to be resolved by the same component. Instead of creating a file for every employee name, you create a single

file with a parameterized name. Next.js uses `[employee].js`, whereas Remix uses `$employee.js`. Then you'd be able to extract the name of the employee from the file by using the parameter in the filename. You can do the same thing with folders too, so you can have a `[country]/` or `$country/` folder. Table 11.2 shows some examples.

Table 11.2 Dynamic routing in a website framework includes parameterized URLs inside the regular location.

	Component location		
URL	**Next.js**	**Remix v1**	**Remix v2**
`http://localhost`	`/pages`	`/app/routes`	`/app/routes`

`/user/1`
`/user/2`
`/user/...`
	`/user`
	`/[id].jsx`
		`/user`
		`/$id.jsx`
			`/user.$id.jsx`

`/shop/chairs/page/1`
`/shop/tables/page/1`
`/show/tables/page/2`
`/shop/.../page/...`
	`/shop`
	`- /[cat]`
	`- /page`
	`- /[page].jsx`
		`/shop`
		`- /$cat`
		`- /page`
		`- /$page.jsx`
			`/shop.$cat.page.`
			`↪ $page.jsx`

In the last example in table 11.2, the folder structure gets quite complex in both Next.js and Remix v1, whereas in Remix v2, the folder structure is still flat, which is the main reason why Remix changed its routing mechanism. If your website has 30 pages, it can be difficult to get an overview if the files are spread across many folders, often containing only one or two files. With a flat file structure, development is much easier.

There are many more aspects of dynamic and nested routing to consider for complex applications, but for our simple application, we have all we need. When we get to the frameworks, we'll see that both solve our needs elegantly.

ROUTING IN OUR WEATHER APP

In our weather app, we need a root route for the front page, a route for the country page, and a route for the city page. All in all, these requirements result in the component locations in table 11.3. Note that we will be using Remix v2 routing for our application.

Table 11.3 The three routes required in our weather application and their file locations inside Next.js, Remix v1, and Remix v2

	Component location		
URL	**Next.js**	**Remix v1**	**Remix v2**
`http://localhost`	`/pages`	`/app/routes`	`/app/routes`
`/`	`/index.jsx`	`/index.jsx`	`/_index.jsx`
`/{country}`	`/[cc]/index.jsx`	`/$cc/index.jsx`	`/$cc.jsx`
`/{country}/{city}`	`/[cc]/[city].jsx`	`/$cc/$city.jsx`	`/$cc.$city.jsx`

11.3.2 Using data in a route

Now we know where to place our components, but what do we put in them? How do we load data from the database and from external services and put that data in our React components?

This next part is one of the main purposes of using a React website framework: you want to be able to preload your HTML from the server, already filled with all the relevant data required to display the page even before React starts doing its magic in the browser.

Earlier, I described the full page as two pieces of a simple puzzle. Each framework creates this puzzle slightly differently, but it will be obvious that these puzzles are both simple and easy to modify.

DEFINING ROUTE DATA IN NEXT.JS

Routes in Next.js are React components defined in route files. Route files are files inside the pages folder, and each page must export a default React component to be a route that can be rendered in the browser.

If you also want a data package to go with that route, define it in a function named `getServerSideProps` and plunk it next to the component. First, let's sketch it in a diagram (figure 11.12).

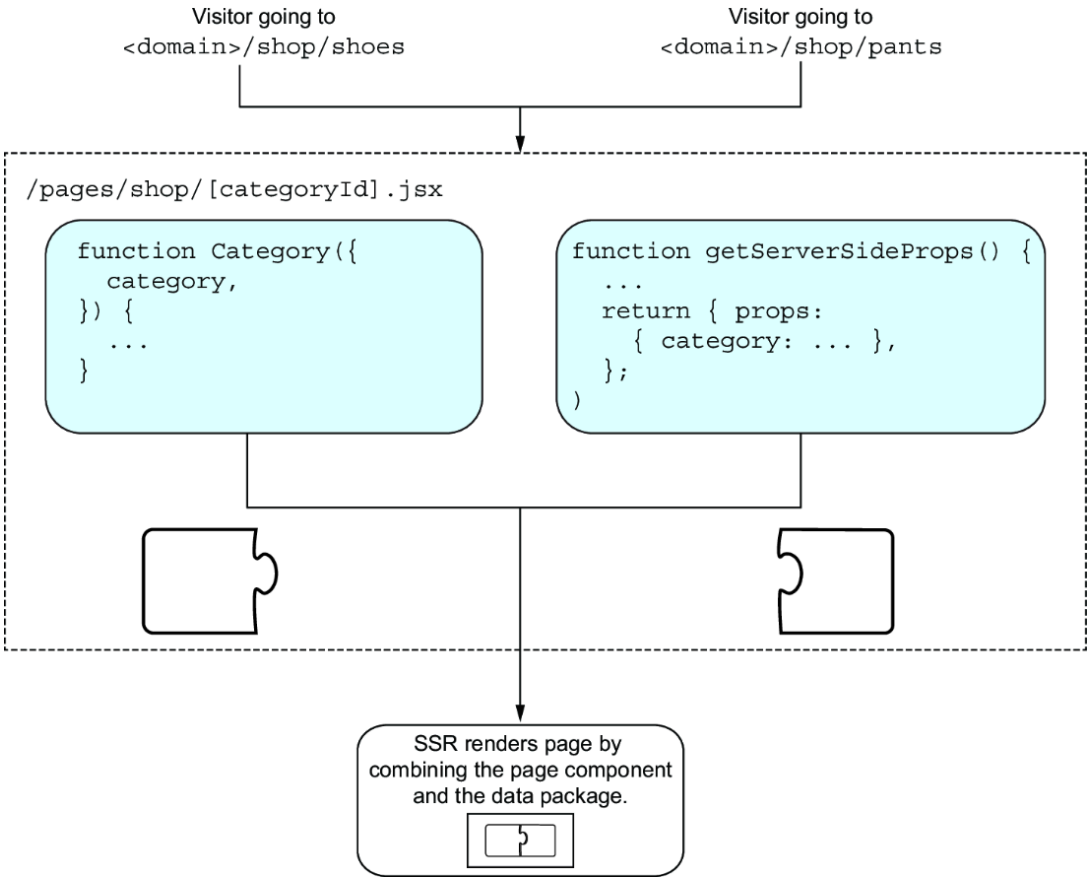


Figure 11.12 For a fictional website, we have shop categories with the URLs `/shop/shoes` and `/ shop/pants`. We dynamically load the category information from the `categoryId` in the URL, and pass that information into the React component. We have to match the same variable name in the return of the former and the property of the latter.

Let's see how this code would implement figure 11.12:

```
// This is the code in the file    #1
// /pages/shop/[categoryId].jsx   #1
export async function
↪getServerSideProps(context) {    #2
  const { categoryId } = context.query;   #3
  const category =     #4
    await getCategoryData(categoryId);    #4
  if (!category) {
    return {     #5
      redirect: {    #5
        destination: "/",   #5
        permanent: false,    #5
      },   #5
    };    #5
  }
  return { props: { category } };    #6
}
function Category({ category }) {   #7
  return (   #7
    <h1>This is the page for the {category.name} category</h1>
  );
}
export default Category; #8
```

#1 Our two functions go in the same route file.

#2 The data package is defined by this function, which takes a context parameter and must return the props for the component to render. It is often asynchronous because of network tasks.

#3 We can get parameters from the URL through the context.

#4 We invoke whatever function we need to retrieve the data from storage or database.

#5 If something fails (such as a bad categoryId), we can return a redirect.

#6 When things go well, we return an object with a props property.

#7 We have to make sure to accept the same props in our component that the data package returns.

#8 Note that the component is a default export, whereas the data package function is a named export.

For our specific application, we're going to need to put some extra things in the data package before we can implement it. We have to load cookies too.

DEFINING ROUTE DATA IN REMIX

Let's do the same thing in Remix that we did in Next.js. First, let's illustrate it in an updated diagram (figure 11.13).

If we implement this same example in code, we'll see many of the same ideas and concepts but with slightly different names:

```
// This is the code in the file    #1
// /app/routes/shop.$categoryId.jsx   #1
export async function loader(context) {     #2
  const { categoryId } = context.params;     #3
  const category = await getCategoryData(categoryId);
  if (!category) {
    redirect("/");    #4
  }
  return json({ category });     #5
}
function Category() {
  const { category } = useLoaderData();     #6
  return (
    <h1>This is the page for the {category.name} category</h1>
  );
}
export default Category;
```

#1 Remember the different naming for parameterized URLs.

#2 The data package is defined in a loader function.

#3 The URL parameters are retrieved from the params property on the context.

#4 Redirects are a bit simpler.

#5 We return JSON using the json() function.

#6 Finally, we read the data package by using a hook inside the component rather than direct props.

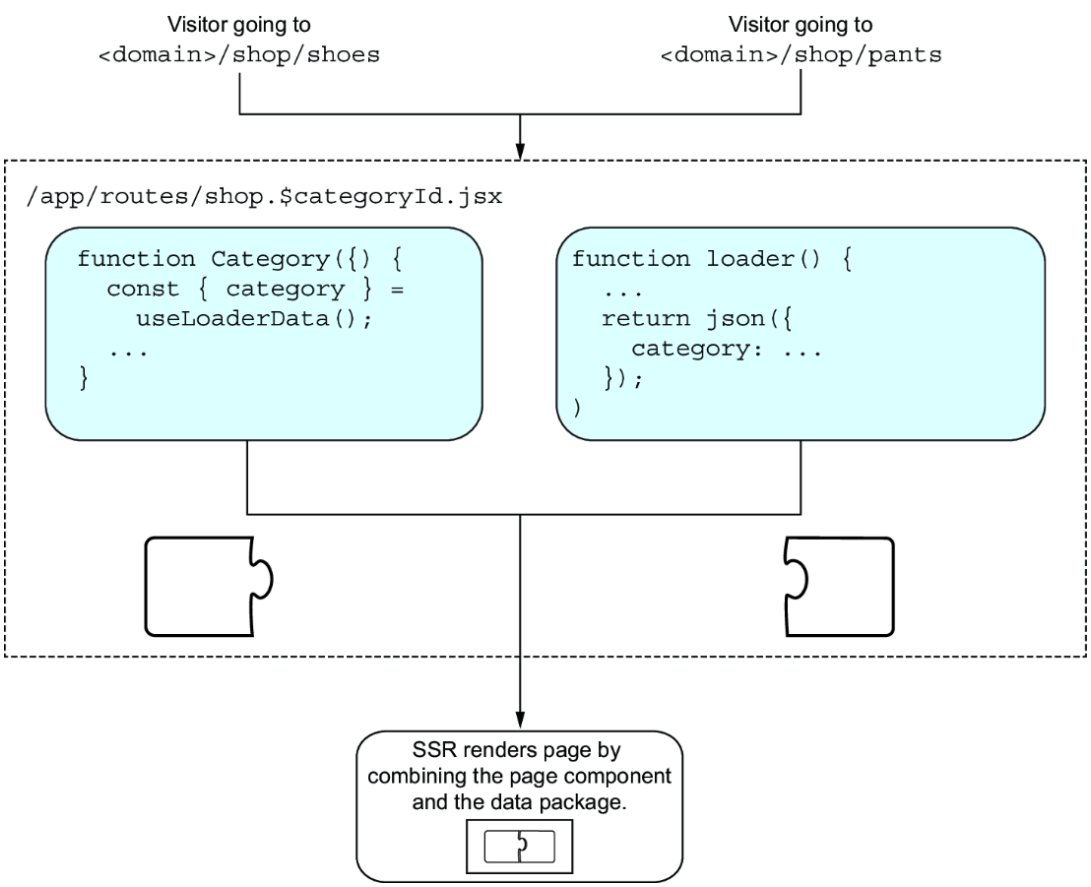


Figure 11.13 In Remix, we name the file slightly differently, define the data package slightly differently, and retrieve the data package values slightly differently. Otherwise, the concept is much like what we did Next.js. There are only so many ways to skin a cat, after all.

Again, we need to understand cookies before we see how to implement data fetching for our particular application.

11.3.3 Storing local data

Everyone in the world can agree on one thing: nobody agrees on everything. Thus, a weather app will always have the problem of displaying temperatures in the wrong scale, so users need a way to toggle between degrees Celsius and degrees Fahrenheit. Yes, I know that there are other scales, but I'll stick to those two. Feel free to implement additional scales in your own implementation.

Which scale you prefer is something you need to store locally, but it would be a bit much to require you to create a user and select a password just to store this tiny bit of info. Instead, we store it locally in your browser so that you can access it when you visit the website again. For

this purpose, we've been using `localStorage` throughout this book, but `localStorage` works only in the browser, not on the server. What's the problem? Well, perfect hydration stops us again. Our server has to render the temperature in one of the two scales, and if the client detects a local browser setting indicating a different scale, hydration fails (figure 11.14).

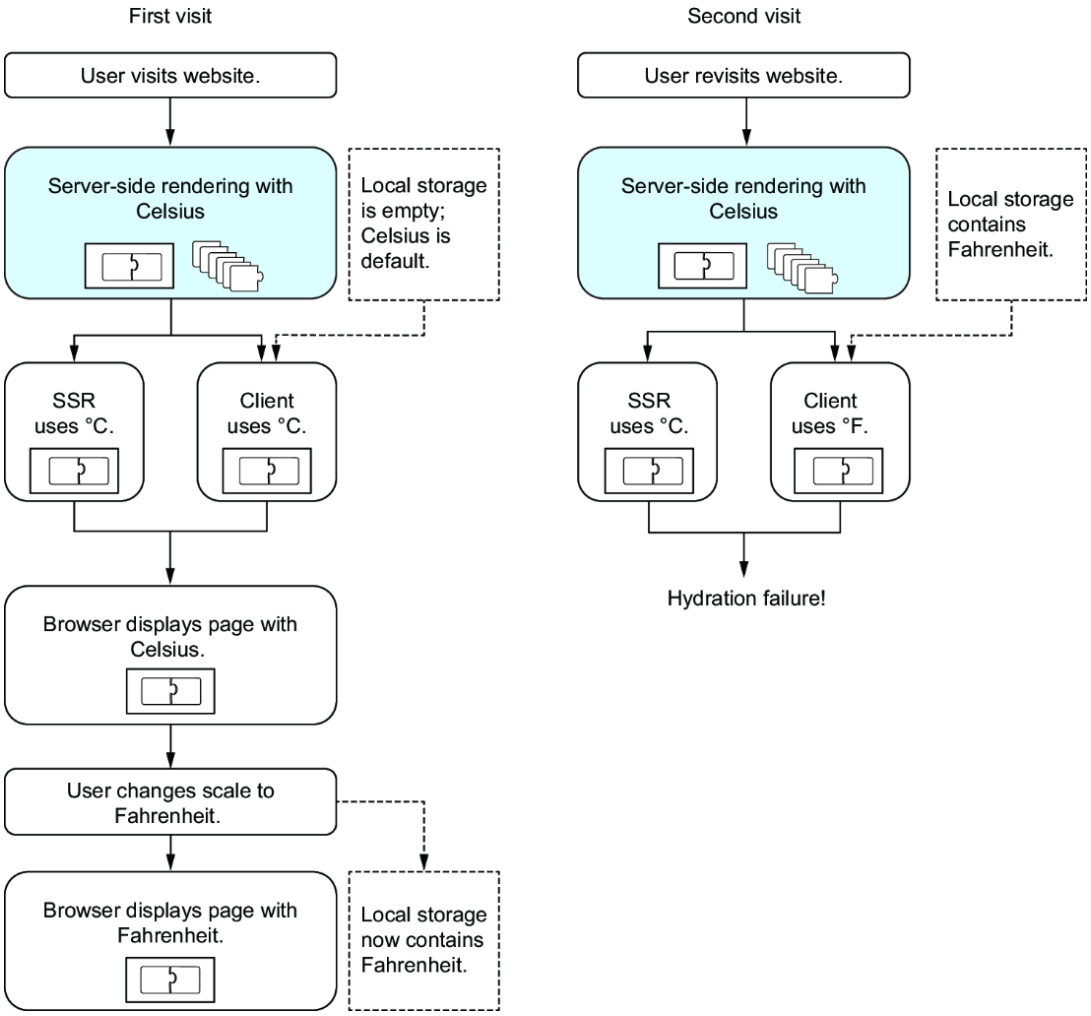


Figure 11.14 The server renders the temperature in Celsius, but local storage in the browser causes the client to render the temperature in Fahrenheit, and perfect hydration fails.

Instead, we can use cookies. Cookies work on both the client and the server, as they are sent to and from the server in every request. Sending data in every request does seem a bit wasteful, which is why you're limited to fewer bytes than you can store in `localStorage`. Our examples that store to-do data in `localStorage` could probably not have been implemented with cookies, as we would have run out of room quickly. With cookies, you get the flow you see in figure 11.15.

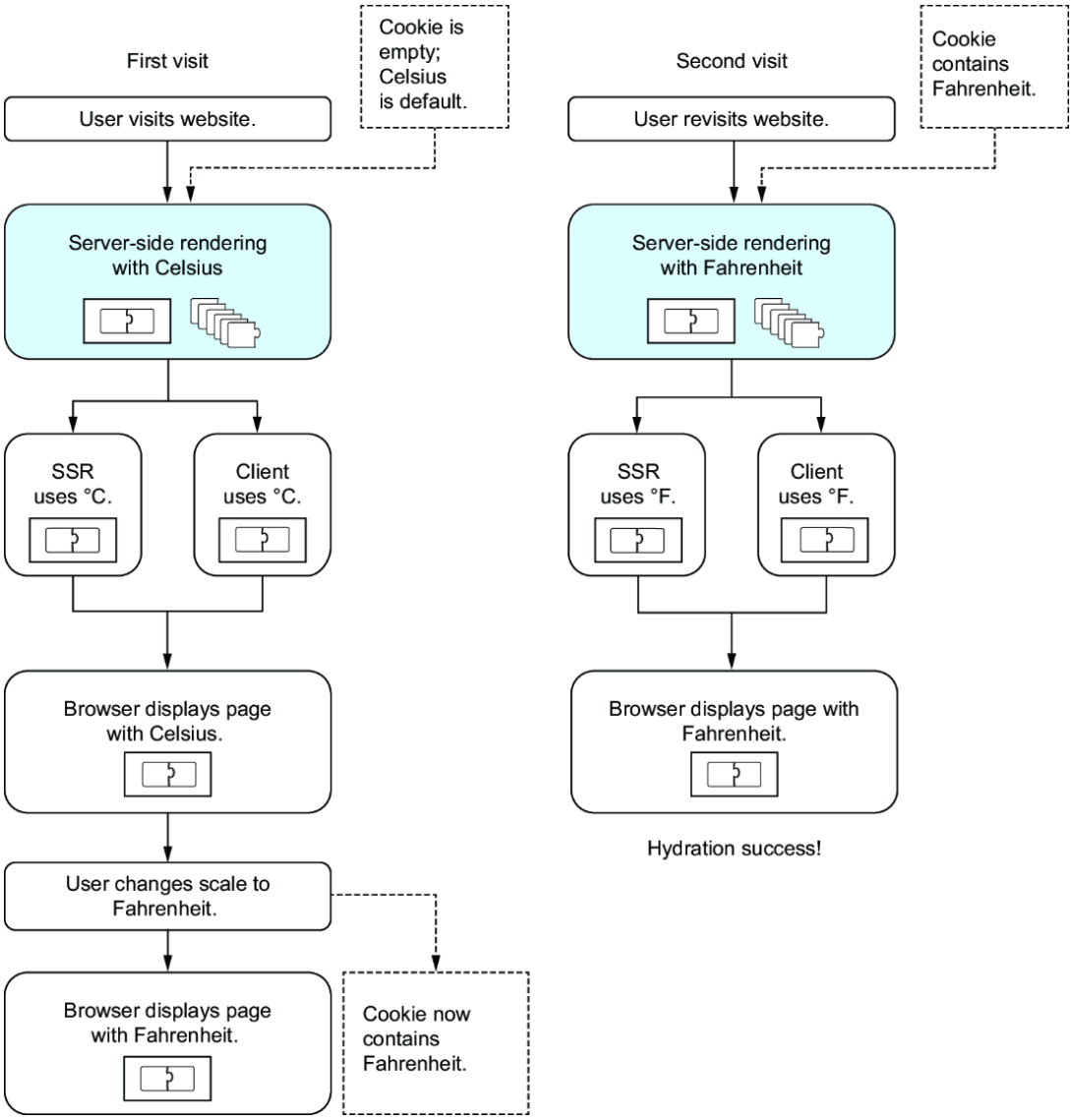


Figure 11.15 With cookies, we can update the local setting on the client and make sure that when the page reloads, the server will also know the correct scale to use, so hydration will happen perfectly.

Cookies can be set, read, edited, and deleted on both the client and the server, so they're perfect for our example. Setting cookies manually in either setting is a bit cumbersome, but fortunately, both of the frameworks we're using come with great tools for manipulating cookies. In Remix, this feature is built into the core, whereas in Next.js, a utility library called `cookies-next` does the job perfectly.

I should mention that we are going to use cookies for another thing: we need to store the user's local position to make it easy for them to find their local weather from the front page. I won't detail how we implement this feature, so check the source code for details.

COOKIES IN NEXT.JS

In Next.js, cookies are implemented by a small third-party module called `cookies-next`. With that module installed (using npm or Yarn as usual), we can read and write cookies on both the client and the server by using a similar (but not identical) API. On the server, you need access to the request object to read cookies, and you need to return the correct header to set a cookie, but on the client, you can read and set a cookie from anywhere, and the browser API will allow the function to work correctly. In our application, we will set cookies only in the browser and read cookies only on the server, which makes things a bit easier. Let's put the proper API calls inside the flow described in figure 11.15, which becomes figure 11.16.

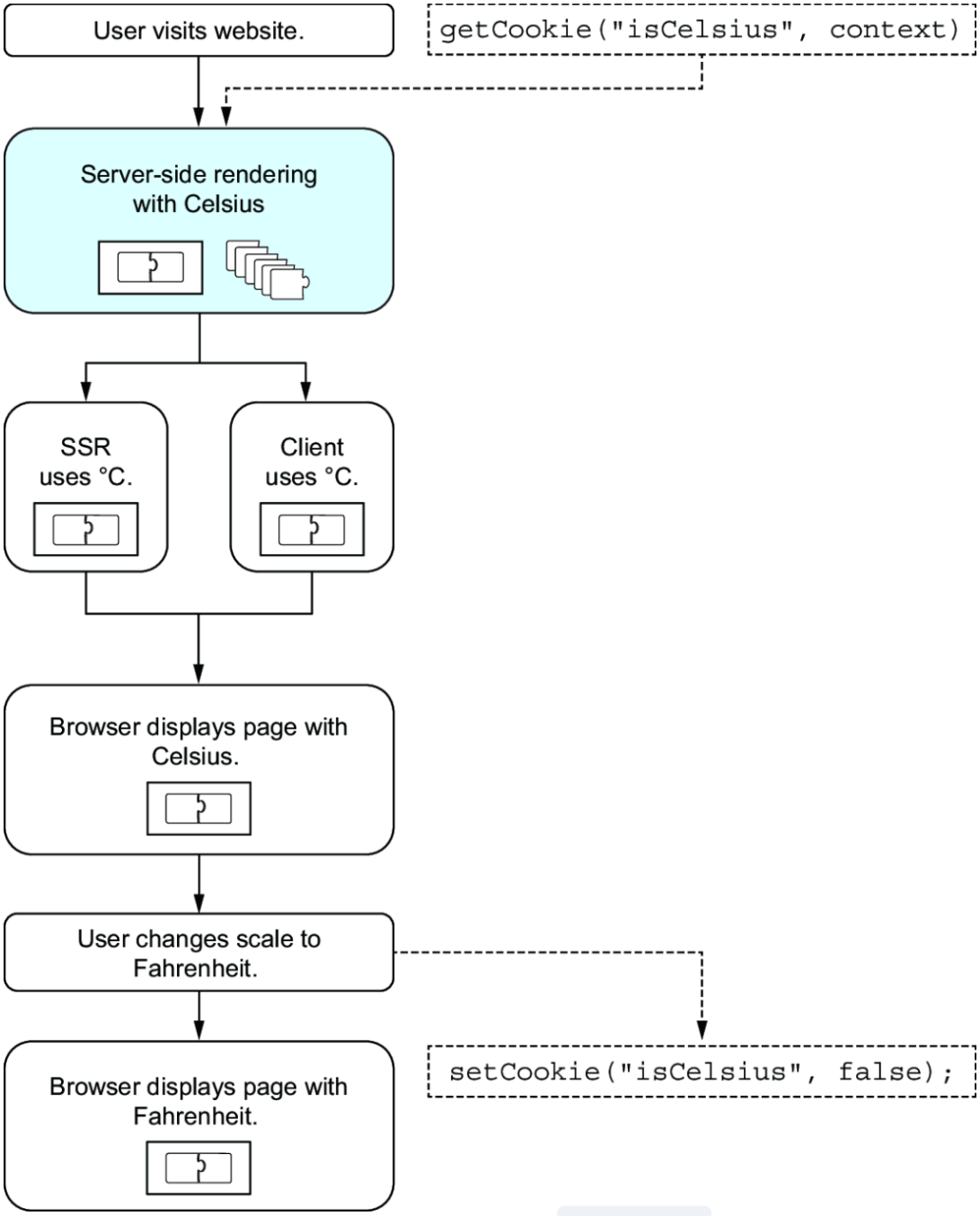


Figure 11.16 We set a cookie by calling `setCookie`, and we read a cookie by calling `getCookie`. (Big-brained stuff here.) Because we want to read the cookie on the server, however, we also pass the context to it.

We use cookies as shown in figure 11.16 for both the temperature (toggling `isCelsius`) and the time format (toggling `hour12`). Putting everything together with the general data loading explained in section 11.3.2, the full city route file, `/pages/[cc]/[city].js`, becomes listing 11.1.

Listing 11.1 City route in Next.js

```
import { getCityData } from "../../services/data";
import { getDefaultFormats } from "../../services/cookies";
import { FormatProvider } from "../../format";
import { CityDisplay } from "../../components";
export async function getServerSideProps(context) {
  const { cc, city } = context.query;
  const data =     #1
    await getCityData(cc.toUpperCase(), city);   #1
  const defaultFormats =    #2
    getDefaultFormats(context);    #2
  if (data.slug !== city) {     #3
    // Redirect, as city was resolved under a different name
    return {
      redirect: {
        destination: `/${cc}/${data.slug}/`,
        permanent: false,
      },
    };

  }
  return { props: { data, defaultFormats } };     #4
}

function City({ data, defaultFormats }) {    #5
  return (
    <FormatProvider     #6
      defaultFormats={defaultFormats}    #6
    >    #6
      <CityDisplay data={data} />     #7
    </FormatProvider>
  );
}
export default City;
```

#1 First, we load the relevant data from the server.

#2 Next, we load more data from cookies by using a utility function because we need to do the same thing in multiple routes.

#3 We have to handle some error cases for the city database result here.

#4 The data package concludes by returning the two relevant bits of information: the city data and the format configuration.

#5 The component starts with those same two pieces of info that the data

package contains.

#6 The page is wrapped in a format provider with the relevant configuration.

#7 The main display component uses the city data to render the page.

The `getDefaultFormats` function is moved to a separate file because we need the functionality in multiple routes. This function is defined in the file `/services/cookies.js`, which you see in listing 11.2.

Listing 11.2 Cookie utils

```
import { getCookie } from "cookies-next";
export function getDefaultFormats(context) {
  const isCelsius = getCookie("nextjs-isCelsius", context);
  const hour12 = getCookie("nextjs-hour12", context);
  return {
    isCelsius: isCelsius !== "false",     #1
    hour12: hour12 === "true",     #2
  };
}
```

#1 The temperature defaults to Celsius.

#2 The time defaults to 24-hour format.

The two other routes, front page and country page, use similar logic and components to render their contents. Check them out in the example.

COOKIES IN REMIX

Cookies in Remix are slightly more complex, and we need a bit of extra context to fully understand why we have to do things differently. In Remix, cookies are encrypted, which sounds good but has consequences. To set or read the cookie, we need a special secret key. If we put that key in the browser, anyone can see it, so the encryption is null and void. Thus, we can read and set cookies only on the server.

ENCRYPTED SERVER-ONLY COOKIES

Using encrypted cookies might seem weird but is a common choice in web development. A cookie can be read by any script on a page. It's not uncommon for a web page to include external scripts, such as analytics tools or various trackers, and you don't want them to access possibly private data. You could store the preferred shipping address or the saved login email in a cookie, and you don't want others to be able to spy on this data. That's why Remix uses encrypted cookies, which become server-only cookies.

We could circumvent this built-in functionality and interact with the cookies as normal by using the HTTP headers, because we don't need the protection of encryption in this case, but we will stick to the framework and its best practices for now.

If we can read and set cookies only on the server, we need to send the information we want to store to the server before it will be stored in a cookie. This approach aligns perfectly with Remix's stated goal of being less dependent on JavaScript. We can create the application in such a way that we can use cookies without JavaScript because we round-trip to the server every time. If the user has JavaScript, however, we'll make the round trip a lot faster courtesy of Remix magic (figure 11.17).

But sending cookie data to the server means that we need to handle this incoming data in Remix. When the form submits and the data is sent to the route by either the browser or Remix, we need to handle that data. For that purpose, we need a third item in the route besides the component and the data package. We also need a data handler. Data handlers, which are integral to Remix, are the opposite of the data package function, `loader()`. The data handler function is called `action()`.

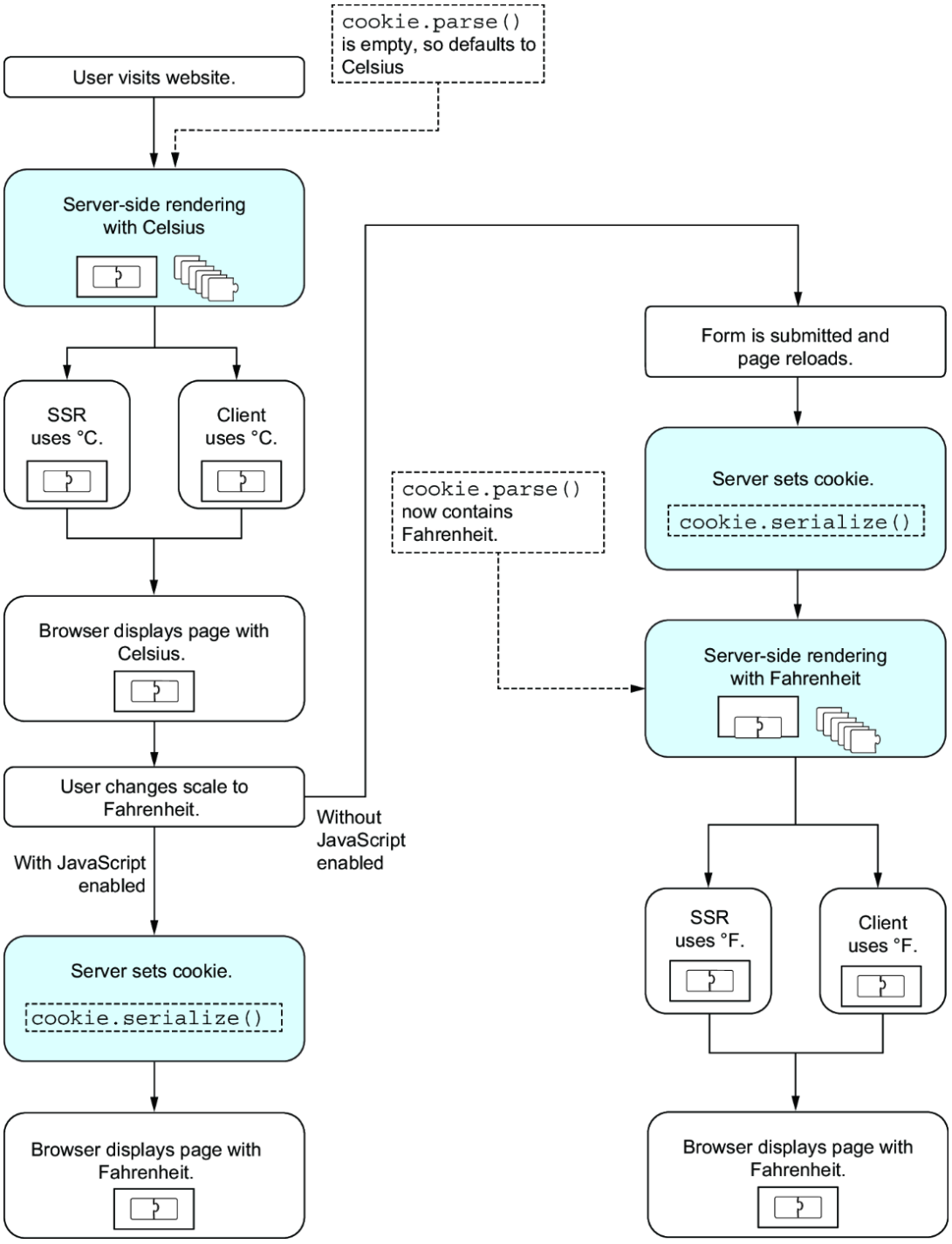


Figure 11.17 The API for setting and reading cookies is a little bit less obvious in Remix, using `parse()` to read and `serialize()` to set. The button to change scale is a submit button in a form. If JavaScript is enabled, Remix kicks in and hijacks the form, sending only data back and forth to the server. If not, the format change still works after the whole page reloads.

All in all, the structure of the Remix route for a city page looks like figure 11.18.

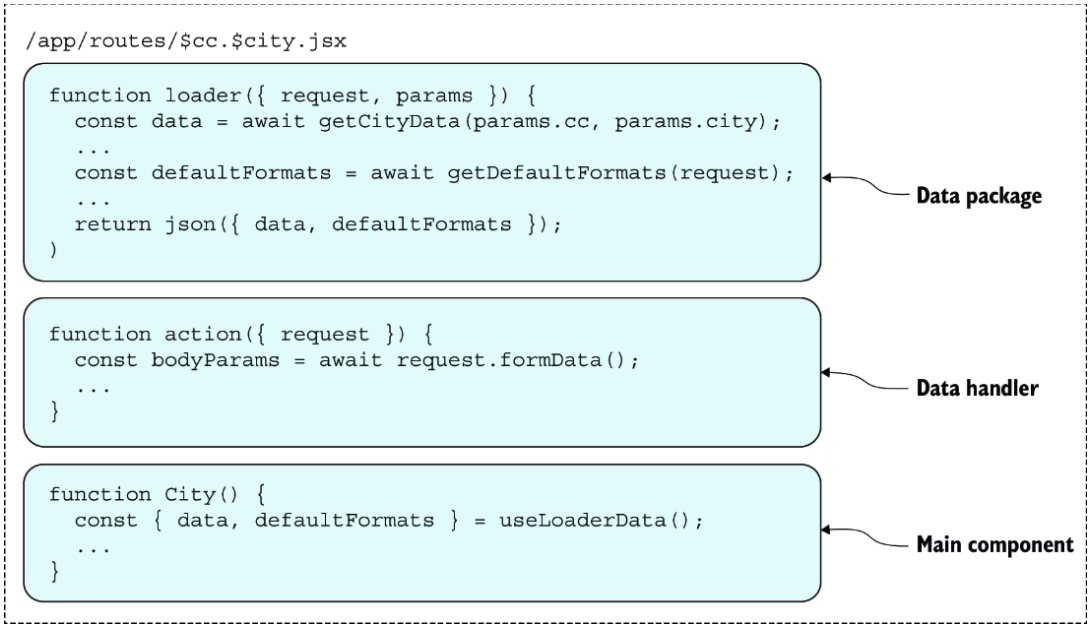


Figure 11.18 The overall structure of the Remix city route contains three parts. Besides the data package and component, which I've covered extensively, we also have a data handler that's capable of handling form submissions.

The following listing shows how we implement the city route.

Listing 11.3 The Remix city route

```
import { useLoaderData } from "@remix-run/react";
import { json, redirect } from "@remix-run/node";
import { getCityData } from "~/services/data";
import { FormatProvider } from "~/format";
import { CityDisplay } from "~/components/cityDisplay";
import {
  getDefaultFormats,
  toggleTemperature,
  toggleTime,
} from "~/services/cookies";
export async function loader({     #1
  request,    #1
  params,    #1
}) {   #1
  const data = await getCityData(params.cc.toUpperCase(), params.city);
  if (!data) {
    redirect("/");
  }
  if (data.slug !== params.city) {
    return redirect(`/${params.cc}/${data.slug}/`);
  }
  const defaultFormats =    #2
    await getDefaultFormats(request);    #2
  return json({ data, defaultFormats });    #3
}
export async function action({ request }) {     #4
  const bodyParams = await request.formData();
  if (bodyParams.has("toggle-temperature")) {     #5
    return toggleTemperature(request);    #5
  }    #5
  if (bodyParams.has("toggle-time")) {     #6
    return toggleTime(request);   #6
  }    #6
}
export default function City() {
  const { data, defaultFormats } =    #7
    useLoaderData();    #7
  return (
    <FormatProvider defaultFormats={defaultFormats}>
      <CityDisplay data={data} />
    </FormatProvider>
```

```
    );
  }
```

#1 The loader function in Remix is fairly identical to the loader function in Next.js but with a Remix-specific API.
#2 We load the default formats by using a utility function as before. Note that we have to await it, as parsing cookies is an asynchronous operation in Remix (unlike Next.js).
#3 The data package contains city data and format information.
#4 Next is the data handler function.
#5 We check to see whether this request is an attempt to toggle the temperature, and if so, we do that.
#6 If it is an attempt to toggle the time format, we do that. Note that we return whatever the toggle util returns.
#7 The component works as normal. We load the data package by using the Remix-provided hook for that purpose.

We have moved some cookie utils to a helper function in `/app/services/cookies`, and you can see an excerpt in the following listing (with some parts that aren't relevant to the city route hidden).

Listing 11.4 Remix cookie utils (excerpt)

```
import { redirect } from "@remix-run/node";
import { temperatureCookie, timeCookie } from "~/cookies.server";
function setCookie(request, value) {    #1
  return redirect(    #1
    request.headers.get("Referer"),    #1
    { headers: { "Set-Cookie": value } },    #1
  );    #1
}    #1
export async function getDefaultFormats(request) {
  const cookieHeader =     #2
    request.headers.get("Cookie") || "";   #2
  const isCelsius = (     #3
    await temperatureCookie.parse(cookieHeader)    #3
  ) !== false;    #3
  const hour12 = (
    await timeCookie.parse(cookieHeader)
  ) !== false;
  return { isCelsius, hour12 };
}
export async function toggleTemperature(request) {
  const { isCelsius } =     #4
    await getDefaultFormats(request);   #4
  return setCookie(    #4
    request,   #4
    await temperatureCookie.serialize(!isCelsius),    #4
  );    #4
}   #4
export async function toggleTime(request) {   #4
  const { hour12 } =    #4
    await getDefaultFormats(request);    #4
  return setCookie(    #4
    request,     #4
    await timeCookie.serialize(!hour12)    #4
  );    #4
}
```

#1 We need a little helper function to return a response that updates a cookie. This function, unfortunately, is not built into Remix.

#2 When reading cookies, we first have to extract the cookie header from the request.

#3 Then we parse the cookie header for each cookie and check the result. Note

that we can have nonstring values in the parsed cookie.

#4 When we toggle one of these format cookies, we first retrieve the current value and then return a response updating the cookie to the opposite value.

Finally, we have the cookie definitions, which are located in `/app/cookies.server.js` per Remix conventions. This file is a very simple, as you see in the following listing.

Listing 11.5 Remix cookie definitions

```
import { createCookie } from "@remix-run/node";
const maxAge = 86_400 * 365; // one year
export const temperatureCookie =    #1
  createCookie("temperature", { maxAge });   #1
export const timeCookie =   #1
  createCookie("time", { maxAge });    #1
export const myCityCookie =   #1
  createCookie("myCity", { maxAge });  #1
```

#1 We define three cookies in the same way but with different names. They're all set to expire after a year.

TIP The Remix cookie is a bit more complex to use and has a slightly unexpected API. Please refer to the Remix documentation for full details on how to use it: **https://remix.run/docs/en/main/utils/cookies**.

11.3.4 Creating an API

Lastly, I want to demonstrate how to create an API in the framework. Although you can create regular data update flows as described in section 11.3.3, which is perfect for forms and similar interactive data, sometimes you need an API that you can invoke from JavaScript and resolve in the client. You may want to manipulate the data by using some client side–only functionality or some external library. If you want to display things in an interactive Google Map instance, you can do so only in a client side setting.

Creating an API is easy in both frameworks, as you'll see when we get to implementations. We need an API to convert coordinates to city names because when we ask the browser for the current location of the visitor, we get only coordinates back. We want to redirect the user to

the correct page, but we don't know the names of the city or country from the two numbers, so we use a third-party API to look them up. This functionality is built into the weather service that we're using. The process of converting coordinates to their real-name counterparts is known as *geocoding*. We place this API route in `/api/geocode`.

API ROUTES IN NEXT.JS

In Next.js, an API route is configured differently from a normal component route. Rather than return a component from the route file, you return a handler function, which is similar to a handler function in Express.js (the underlying server library that powers the Next.js HTTP functionality).

A handler function takes a request and a response as arguments and returns via the response object by setting the status code and response body directly on that object. Following is an example for an API route that returns the string `"Hello world!"`:

```
export default function handler(req, res) {
  res.status(200).json("Hello world!");
}
```

Note that we set the response code and the response JSON object directly rather than return something. This function can be asynchronous as well if we need to wait for data to load. Our API requires us to do three things:

- Read coordinates (latitude and longitude) from the request query.
- Geocode the coordinates via a third-party API.
- Return the response to the client or an error if nothing is found.

We can implement the geocoding API in Next.js.

Listing 11.6 Geocoding API in Next.js

```
import slugify from "slugify";
import { getCityForCoordinate } from "../../services/api";
import { getCountryName } from "../../services/data";
export default async function handler(req, res) {
  const { lat, lng } = req.query;   #1
  const { cityName, countryCode } =     #2
    await getCityForCoordinate(lat, lng);    #2
  if (!cityName || !countryCode) {     #3
    res.status(404).json({ result: "error" });   #3
    return;    #3
  }    #3
  const country = await getCountryName(countryCode);
  const path = `/${slugify(countryCode)}/${slugify(cityName)}`;
  res.status(200)     #4
    .json({ path, country, city: cityName });   #4
}
```

#1 First, we retrieve the coordinates from the query string.

#2 Then, we use the OpenWeather API to look up the location name.

#3 If it fails, we return an error. (The coordinate is probably lost at sea.)

#4 If all goes well, we return a successful response.

API ROUTES IN REMIX

In Remix, API routes aren't special. They're the same as regular component-based routes; they just don't have a component. So if you define only a data package in a route file, it is automatically an API route, which is convenient because we already know how to define a Remix data package. That same example with a route returning the string `"Hello world!"` becomes

```
import { json } from "@remix-run/node";
export function loader({ request }) {
  return json("Hello world!");
}
```

The implementation of our application API route is located in `/app/routes/api.geocode.js`.

Listing 11.7 Geocoding API in Remix

```
import { json, Response } from "@remix-run/node";
import slugify from "slugify";
import { getCityForCoordinate } from "../../services/api";
import { getCountryName } from "../../services/data";
export async function loader({ request }) {
  const url = new URL(request.url);     #1
  const lat = url.searchParams.get("lat");   #1
  const lng = url.searchParams.get("lng");    #1
  const { cityName, countryCode } =    #2
    await getCityForCoordinate(lat, lng);   #2
  if (!cityName || !countryCode) {    #3
    throw new Response(  #3
      "Not Found",  #3
      { status: 404 },   #3
    );   #3
  }    #3
  const country = await getCountryName(countryCode);
  const path = `/${slugify(countryCode)}/${slugify(cityName)}`;
  return json({ path, country, city: cityName });     #4
}
```

#1 First, we retrieve the coordinates from the query string.

#2 Then, we use the OpenWeather API to look up the location name.

#3 If it fails, we return an error. (Maybe the coordinates are in space.)

#4 If all goes well, we return a successful response.

11.4 Alternative React-based website frameworks

Next.js and Remix aren't the only choices, of course; they're the most popular and the most hyped, respectively. Remix is fairly new, so if you read this book a couple of years down the line, it may have been scrapped, for all I know. Next.js is much more likely to still be around, as it has been around for quite a while.

Other libraries try to market themselves within a niche of the web application market rather than tackle all web applications, like the frameworks I've detailed in this chapter. Some noteworthy alternatives include

- *Gatsby* (**https://www.gatsbyjs.com**)—This framework is best for small websites and can even compile your React components down to a pure HTML-and-CSS-only website with no JavaScript in sight.
- *Astro* (**https://astro.build**)—Astro is another fairly new project that's ideal for content-heavy websites rather than complex web applications. It's ideal for marketing websites, documentation, blogs, news sites, and the like. Astro is not React specific and can work with other libraries.
- *RedwoodJS* (**https://redwoodjs.com**)—This framework is good for fast prototyping if you're building a new startup and want to get going quickly without worrying much about customization and tailoring. With a few lines of script, you have a basic website and content management system (CMS) up and running. This CMS includes forms and tables to list, display, create, and edit your objects in the database, and it comes with many great tools built in.
- *Qwik* (**https://qwik.builder.io**)—This framework is hyperfocused on performance and speed, so if delivering your website in milliseconds is most important, such as for a web shop or a campaign landing page, Qwik may be your best choice. By default, Qwik uses its own simpler, scaled-down variant of React, but you can use regular React within it.

Note that all these alternatives have found a niche, whereas Next.js targets any type of website. The reason is mostly that Next.js has been king of React website frameworks for a long time, and most direct competitors have drowned in its wake.

Only Remix dared take the reigning champion on directly, and we've yet to see who will emerge as the victor. Remix and Next.js may end up splitting the market, but for now, these two are the main contenders for the general React website framework throne. Next.js and Remix are far from alone, however, so always check around to see whether some other framework fits your use case better.

Summary

- React website frameworks are powerful ways to build React websites or even regular HTML websites.

- React website frameworks allow you to use React as a natural part of a JavaScript- or TypeScript-based fullstack development environment, in which React serves as both the templating engine for server-side rendered HTML and the front end UI when the data has been delivered to the browser.
- Many React website frameworks are optimized for extremely fast performance, and this optimization alone is reason enough to consider using them.
- The performance gain from React website frameworks comes primarily from SSR. By pregenerating the HTML on the server, the browser can show the resulting page in milliseconds after the visitor arrives rather than after React loads and kicks in. To get this performance gain in a React website framework with SSR, however, you need perfect hydration. When React hydrates your page, React expects the browser-rendered HTML to be identical to the server-rendered HTML. If hydration fails, a significant performance penalty overshadows the initial performance gain.
- Next.js and Remix are two of the most popular frameworks for rendering React on the server. Both frameworks come with a slew of other web development libraries as recommendations: styling, data management, database access, and so on.
- Using external APIs in a React website framework is a breeze because we can use the full functionality of JavaScript in Node.js as it runs on the server and render the resulting output only in the browser, using the less-capable browser JavaScript engine.
- Routes are essential in both Next.js and Remix. When defining routes, you can use tools to define more or less complex routes, including dynamic data loading, data handlers, and parameterized URLs.
- Local storage is a no-go for SSR because it is not available on the server. Using cookies is often a reasonable alternative as long as the data items are relatively small. Using cookies in either framework is fairly easy, but Remix has a significant amount of extra overhead due to the slightly more complicated implementation.
- Remix and Next.js aren't the only libraries available, so do some research for your particular project. You might find some other framework that's tailored perfectly to your use case.