SAMPLE CHAPTER

React IN ACTION

Mark Tielens Thomas

MANNING

React in Action

by Mark Tielens Thomas

Chapter 1

brief contents

1 Meet React

This chapter covers

- Introducing React
- Some of React's high-level concepts and paradigms
- The virtual DOM
- Components in React
- React for teams
- Tradeoffs of using React

If you work as a web engineer in the tech industry, chances are you've heard of React. Maybe it was somewhere online like Twitter or Reddit. Maybe a friend or colleague mentioned it to you or you heard a talk about it at a meetup. Wherever it was, I bet that what you heard was probably either glowing or a bit skeptical. Most people tend to have a strong opinion about technologies like React. Influential and impactful technologies tend to generate that kind of response. For these technologies, often a smaller number of people initially "get it" before the technology catches on and moves to a broader audience. React started this way, but now enjoys immense popularity and use in the web engineering world. And it's popular for

good reason: it has a lot to offer and can reinvigorate, renew, or even transform how you think about and build user interfaces.

1.1 Meet React

React is a JavaScript library for building user interfaces across a variety of platforms. React gives you a powerful mental model to work with and helps you build user interfaces in a declarative and component-driven way. We'll unpack these ideas and much more over the course of the book, but that's what React is in the broadest, briefest sense.

Where does React fit into the broader world of web engineering? You'll often hear React talked about in the same space as projects like Vue, Preact, Angular, Ember, Webpack, Redux and other well-known JavaScript libraries and frameworks. React is often a major part of front-end applications and shares similar features with the other libraries and frameworks just mentioned. In fact, many popular front-end technologies are more like React in subtle ways now than in the past. There was a time when React's approach was novel, but other technologies have since been influenced by React's component-driven, declarative approach. React continues to maintain a spirit of rethinking established best practices, with the main goal being providing developers with an expressive mental model and a performant technology to build UI applications.

What makes React's mental model powerful? It draws on deep areas of computer science and software engineering techniques. React's mental model draws broadly on functional and object-oriented programming concepts and focuses on components as primary units for building with. In React applications, you create interfaces from components. React's rendering system manages these components and keeps the application view in sync for you. Components often correspond to aspects of the user interface, like datepickers, headers, navbars, and others, but they can also take responsibility for things like client-side routing, data formatting, styling, and other responsibilities of a client-side application.

Components in React should be easy to think about and integrate with other React components; they follow a predictable lifecycle, can maintain their own internal state, and work with "regular old JavaScript." We'll dive into these ideas over the course of the rest of the book, but we can look at them at a high level right now. Figure 1.1 gives you an overview of the major ingredients that go into a React application. Let's look at each part briefly:

- *Components*—Encapsulated units of functionality that are the primary unit in React. They utilize data (*properties* and *state*) to render your UI as output; we'll explore how React components work with data later in chapter 2 onward. Certain types of React components also provide a set of lifecycle methods that you can hook into. The *rendering process* (outputting and updating a UI based on your data) is predictable in React, and your components can hook into it using React's APIs.
- *React libraries*—React uses a set of core libraries. The core React library works with the `react-dom` and `react-native` libraries and is focused on component

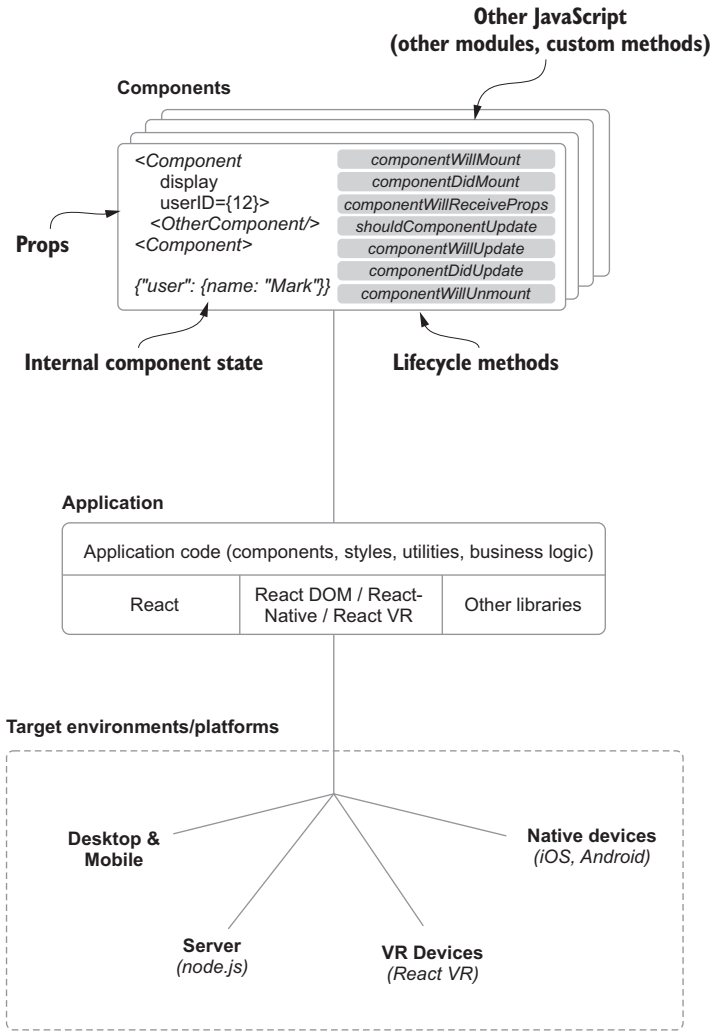

Figure 1.1 React allows you to create user interfaces from components. Components maintain their own state, are written in and work with "vanilla" JavaScript, and inherit a number of helpful APIs from React. Most React apps are written for browser-based environments, but can also be used in native environments like iOS and Android. For more about React Native, see Nader Dabit's *React Native in Action*, also available from Manning.

specification and definition. It allows you to build a tree of components that a renderer for the browser or another platform can use. `react-dom` is one such renderer and is aimed at browser environments and server-side rendering. The React Native libraries focus on native platforms and let you create React applications for iOS, Android, and other platforms.

- *Third-party libraries*—React doesn't come with tools for data modeling, HTTP calls, styling libraries, or other common aspects of a front-end application. This leaves you free to use additional code, modules, or other tools you prefer in your application. And even though these common technologies don't come bundled with React, the broader ecosystem around React is full of incredibly useful libraries. In this book, we'll use a few of these libraries and devote chapters 10 and 11 to looking at Redux, a library for state management.
- *Running a React application*—Your React application runs on the platform you're building for. This book focuses on the web platform and builds a browser and server-based application, but other projects like React Native and React VR open the possibility of your app running on other platforms.

We'll spend lots of time exploring the ins and outs of React in this book, but you may have a few questions before getting started. Is React something for you? Who else is using React? What are some of the tradeoffs of using React or not? These are important questions about a new technology that you'll want answered before adopting it.

1.1.1 Who this book is for

This book is for anyone who's working on or interested in building user interfaces. Really, it's is for anyone who's curious about React, even if you don't work in UI engineering. You'll get the most out of this book if you have some experience with using JavaScript to build front-end applications.

You can learn how to build applications with React as long as you know the basics of JavaScript and have some experience building web applications. I don't cover the fundamentals of JavaScript in this book. Topics like prototypal inheritance, ES2015+ code, type coercion, syntax, keywords, asynchronous coding patterns like async/await, and other fundamental topics are beyond the scope of this book. I do lightly cover anything that's especially pertinent to React but don't dive deep into JavaScript as a language.

This doesn't mean you can't learn React or won't get anything from this book if you don't know JavaScript. But you'll get much more if you take the time to learn JavaScript first. Charging ahead without a working knowledge of JavaScript will make things more difficult. You might run into situations where things might seem like "magic" to you—things will work, but you won't understand why. This usually hurts rather than helps you as a developer, so ... last warning: get comfortable with the basics of JavaScript before learning React. It's a wonderfully expressive and flexible language. You'll love it!

You may already know JavaScript well and may have even dabbled in React before. This wouldn't be too surprising given how popular React has become. If this is you, you'll be able to gain a deeper understanding of some of the core concepts of React. But I don't cover highly specific topics you may be looking for if you've been working

with React for a while. For those, see other React-related Manning titles like *React Native in Action*.

You may not fit into either group and may want a high-level overview of React. This book is for you, too. You'll learn the fundamental concepts of React and you'll have access to a sample application written in React—check out the running app at https://social.react.sh. You'll be able to see the basics of building a React application in practice and how it might be suited to your team or next project.

1.1.2 A note on tooling

If you've worked extensively on front-end applications in the past few years, you won't be surprised by the fact that the tooling around applications has become as much a part of the development process as frameworks and libraries themselves. You're likely using something like Webpack, Babel, or other tools in your applications today. Where do these and other tools fit into this book, and what you need to know?

You don't need to be a master of Webpack, Babel, or other tools to enjoy and read this book. The sample application I've created utilizes a handful of important tools, and you can feel free to read through the configuration code for these in the sample application, but I don't cover these tools in depth in this book. Tooling changes quickly, and more importantly, it would be well outside the scope of this book to cover these topics in depth. I'll be sure to note anywhere tooling is relevant to our discussion, but besides that I'll avoid covering it.

I also feel that tooling can be a distraction when learning a new technology like React. You're already trying to get your head around a new set of concepts and paradigms—why clutter that with learning complex tooling too? That's why chapter 2 focuses on learning "vanilla" React first before moving on to features like JSX and JavaScript language features that require build tools. The one area of tooling that you'll need to be familiar with is npm. npm is the package management tool for JavaScript, and you'll use it to install dependencies for your project and run project commands from the command line. It's likely you're already familiar with npm, but if not, don't let that dissuade you from reading the book. You only need the most basic terminal and npm skills to go forward. You can learn about npm at https://docs.npmjs.com/getting-started/what-is-npm.

1.1.3 Who uses React?

When it comes to open source software, who is (and who isn't) using it is more than just a matter of popularity. It affects the experience you'll have working with the technology (including availability of support, documentation, and security fixes), the level of innovation in the community, and the potential lifetime of a certain tool. It's generally more fun, easier, and overall a smoother experience to work with tools that have a vibrant community, a robust ecosystem, and a diversity of contributor experience and background.

React started as a small project but now has broad popularity and a vibrant community. No community is perfect, and React's isn't either, but as far as open source

communities go, it has many important ingredients for success. What's more, the React community also includes smaller subsets of other open source communities. This can be daunting because the ecosystem can seem vast, but it also makes the community robust and diverse. Figure 1.2 shows a map of the React ecosystem. I mention various libraries and projects throughout the course of the book, but if you're curious to learn more about the React ecosystem, I've put together a guide at https://ifelse.io/react-ecosystem. I'll keep this updated over time and ensure it evolves as the ecosystem does.

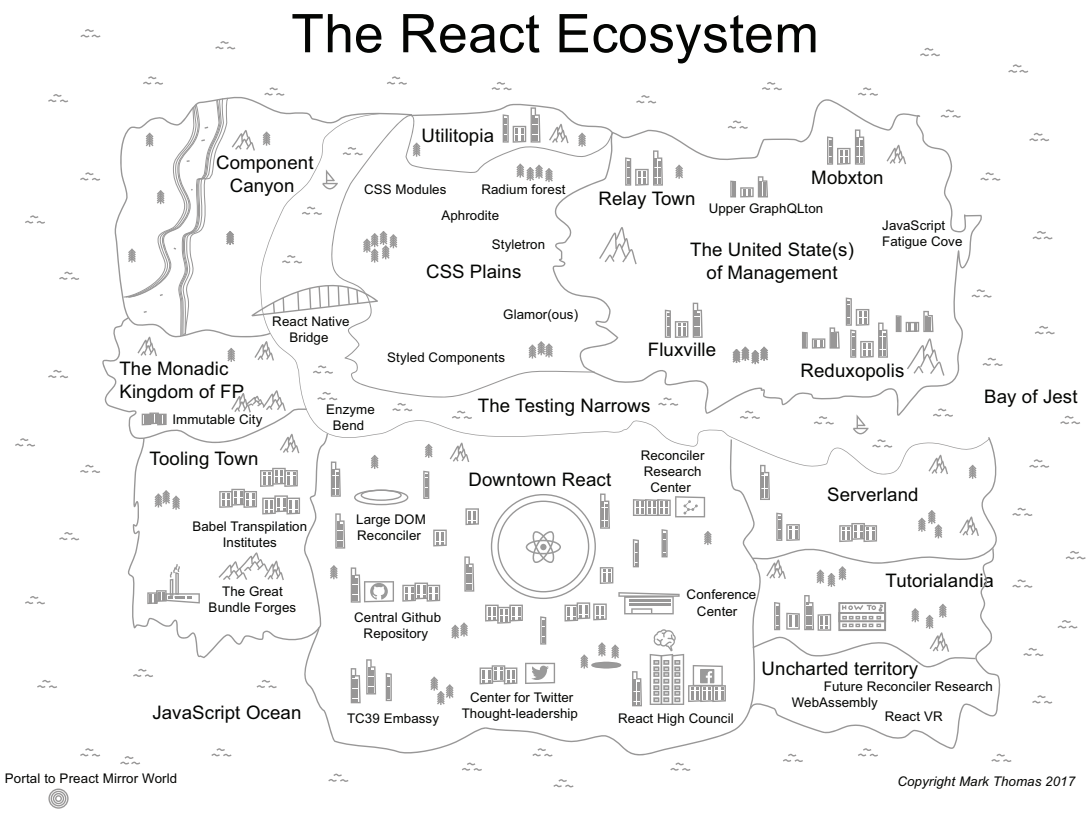

Figure 1.2 A map of the React ecosystem is diverse—even more so than I can represent here. If you'd like to learn more, check out my guide at https://ifelse.io/react-ecosystem, which will help you find your way in the React ecosystem when starting out.

The primary way you might interact with React is probably in open source, but you likely use apps built with it every day. Many companies use React in different and exciting ways. Here are a few of the companies using React to power their products:

- Facebook
- Netflix
- New Relic
- Uber
- Wealthfront
- Heroku
- PayPal
- BBC
- Microsoft
- NFL
- And more!
- Asana
- ESPN
- Walmart
- Venmo
- Codecademy
- Atlassian
- Asana
- Airbnb
- Khan Academy
- FloQast

These companies aren't blindly following the trends of the JavaScript community. They have exceptional engineering demands that impact a huge number of users and must deliver products on hard deadlines. Someone saying, "I heard React was good; we should React-ify everything!" won't fly with managers or other engineers. Companies and developers want good tools that help them think better and move quickly so they can build high-impact, scalable, and reliable applications.

1.2 What does React not do?

So far, I've been talking about React at a high-level: who uses it, who this book is for, and so on. My primary goals in writing this book are to teach you how to build applications with React and empower you as an engineer. React isn't perfect, but it's genuinely been a pleasure to work with, and I've seen teams do great things with it. I love writing about it, building with it, hearing talks about it at conferences, and engaging in the occasional spirited debate about this or that pattern.

But I would be doing you a disservice if I didn't talk about some of the downsides of React and describe what it *doesn't* do. Understanding what something can't do is as important as understanding what it can do. Why? The best engineering decisions and thinking usually happen in terms of tradeoffs instead of opinions or absolutes ("React is fundamentally better than tool *X* because I like it more"). On the former point: you're probably not dealing with two totally different technologies (COBOL versus JavaScript); hopefully you're not even considering technologies that are fundamentally unsuited to the task at hand. And to the latter point: building great projects and solving engineering challenges should never be about opinions. It's not that people's opinions don't matter—that's certainly not true—it's that opinions don't make things work well or at all.

1.2.1 Tradeoffs of React

If tradeoffs are the bread and butter of good software evaluation and discussion, what tradeoffs are there with React? First, React is sometimes called *just the view*. This can be misconstrued or misunderstood because it can lead you to think React is just a templating system like Handlebars or Pug (née Jade) or that it has to be part of an MVC (model-view-controller) architecture. Neither is true. React can be both of those things, but it can be much more. To make things easier, I'll describe React more in terms of what it *is* than what it's not ("just the view," for example). React is a *declarative*, *component-based* library for building user interfaces that works on a variety of platforms: web, native, mobile, server, desktop, and even on virtual reality platforms going forward (React VR).

This leads to our first tradeoff: React is primarily concerned with the *view* aspects of UI. This means it's not built to do many of the jobs of a more comprehensive framework or library. A quick comparison to something like Angular might help drive this point home. In its most recent major release, Angular has much more in common with React than it previously did in terms of concepts and design, but in other ways it covers much more territory than React. Angular includes opinionated solutions for the following:

- HTTP calls
- Form building and validation
- Routing
- String and number formatting
- Internationalization
- Dependency injection
- Basic data modeling primitives
- Custom testing framework (although this isn't as important a distinction as the other areas)
- Service workers included by default (a worker-style approach to executing JavaScript)

That's a lot, and in my experience there are generally two ways people tend to react[1] to all these features coming with a framework. Either it's along the lines of "Wow, I don't have to deal with all those myself" or it's "Wow, I don't get to choose how I do anything." The upside of frameworks like Angular, Ember, and the like is that there's usually a well-defined way to do things. For example, routing in Angular is done with the built-in Angular Router, HTTP tasks are all done with the built-in HTTP routines, and so on.

There's nothing fundamentally wrong with this approach. I've worked on teams where we used technologies like this and I've worked on teams where we went the

[1] Pun not intended but, hey, it's a book about React, so there it is.

more flexible direction and chose technologies that "did one thing well." We did great work with both kinds of technologies, and they served their purposes well. My personal preference is toward the choose-your-own, does-one-thing-well approach, but that's really neither here nor there; it's all about tradeoffs. React doesn't come with opinionated solutions for HTTP, routing, data modeling (although it certainly has opinions about data flow in your views, which we'll get to), or other things you might see in something like Angular. If your team sees this as something you absolutely can't do without in a singular framework, React might not be your best choice. But in my experience, most teams want the flexibility of React coupled with the mental model and intuitive APIs that it brings.

One upside to the flexible approach of React is that you're free to pick the best tools for the job. Don't like the way *X* HTTP library works? No problem—swap it out for something else. Prefer to do forms in a different way? Implement it, no problem. React provides you with a set of powerful primitives to work with. To be fair, other frameworks like Angular will usually allow you to swap things out too, but the de facto and community-backed way of doing things will usually be whatever is built-in and included.

The obvious downside to having more freedom is that if you're used to a more comprehensive framework like Angular or Ember, you'll need to either come up with or find your own solution for different areas of your application. This can be a good thing or a bad thing, depending on factors like developer experience on your team, engineering management preferences, and other factors specific to your situation. There are plenty of good arguments for the one-size-fits-all as well as the does-one-thing-well approaches. I tend to be more convinced by the approach that lets you adapt and make flexible, case-by-case decisions about tooling over time in a way that entrusts engineering teams with the responsibility to determine or create the right tools. There's also the incredibly broader JavaScript ecosystem to consider—you'll be hard-pressed to find *nothing* aimed at a problem you're solving. But at the end of the day, the fact remains that excellent, high-impact teams use both sorts of approaches (sometimes at the same time!) to build out their products.

I'd be remiss if I didn't mention lock-in before moving on. It's an unavoidable fact that JavaScript frameworks are rarely truly interoperable; you can't usually have an app that's part Angular, part Ember, part Backbone, and part React, at least not without segmenting off each part or tightly controlling how they interact. It doesn't usually make sense to put yourself in that sort of situation when you can avoid it. You usually go with one and maybe temporarily, at most, two primary frameworks for a particular application.

But what happens when you need to change? If you use a tool with wide-ranging responsibilities like Angular, migrating your app is likely going to be a complete rewrite due to the deep idiomatic integration of your framework. You can rewrite smaller parts of the application, but you can't just swap out a few functions and expect everything to work. This is an area where React can shine. It employs relatively few

"magic" idioms. That doesn't mean it makes migration painless, but it does help you to potentially forgo incurring the cost of a tightly integrated framework like Angular if you migrate to or from it.

Another tradeoff you make when choosing React is that it's primarily developed and built by Facebook and is meant to serve the UI needs of Facebook. You might have a hard time working with React if your application is fundamentally different than the UI needs of Facebook's apps. Fortunately, most modern web apps are in React's technological wheelhouse, but there are certainly apps that aren't. These might also include apps that don't work within the conventional UI paradigms of modern web apps or apps that have very specific performance needs (such as a high-speed stock ticker). Yet even these can often be addressed with React, though some situations require more-specific technologies.

One last tradeoff we should discuss is React's implementation and design. Baked into the core of React are systems that handle updating the UI for you when the data in your components change. They execute changes that you can hook into using certain methods called *lifecycle methods*. I cover these extensively in later chapters. React's systems that handle updating your UI make it much easier to focus on building modular, robust components that your application can use. The way React abstracts away most of the work of keeping a UI up-to-date with data is a big part of why developers enjoy working with it so much and why it's a powerful primitive in your hands. But it shouldn't be assumed that there are no downsides or tradeoffs made with respect to the "engines" that power the technology.

React is an abstraction, so the costs of it being an abstraction still remain. You don't get as much visibility into the system you're using because it's built in a particular way and exposed through an API. This also means you'll need to build your UI in an idiomatically React way. Fortunately, React's APIs provide "escape hatches" that let you drop down into lower levels of abstraction. You can still use other tools like jQuery, but you'll need to use them in a React-compatible way. This again is a tradeoff: a simpler mental model at the cost of not being able to do absolutely everything how you'd like.

Not only do you lose some visibility to the underlying system, you also buy into the way that React does things. This tends to impact a narrower slice of your application stack (only views instead of data, special form-building systems, data modeling, and so on), but it affects it nonetheless. My hope is that you'll see that the benefits of React far outweigh the cost of learning it and that the tradeoffs you make when using it generally leave you in a much better place as a developer. But it would be disingenuous for me to pretend that React will magically solve all your engineering challenges.

1.3 The virtual DOM

We've talked a little bit about some of the high-level features of React. I've posited that it can help you and your team become better at creating user interfaces and that part

of this is due to the mental model and APIs that React provides. What's behind all that? A major theme in React is a drive to simplify otherwise complex tasks and abstract unnecessary complexity away from the developer. React tries to do just enough to be performant while freeing you up to think about other aspects of your application. One of the main ways it does that is by encouraging you to be *declarative* instead of *imperative*. You get to declare how your components should behave and look under different states, and React's internal machinery handles the complexity of managing updates, updating the UI to reflect changes, and so on.

One of the major pieces of technology driving this is the virtual DOM. A *virtual DOM* is a data structure or collection of data structures that mimics or mirrors the Document Object Model that exists in browsers. I say *a* virtual DOM because other frameworks such as Ember employ their own implementation of a similar technology. In general, a virtual DOM will serve as an intermediate layer between the application code and the browser DOM. The virtual DOM allows the complexity of change detection and management to be hidden from the developer and moved to a specialized layer of abstraction. In the next sections, we'll look from a high level at how this works in React. Figure 1.3 shows a simplified overview of the DOM and virtual DOM relationship that we'll explore shortly.

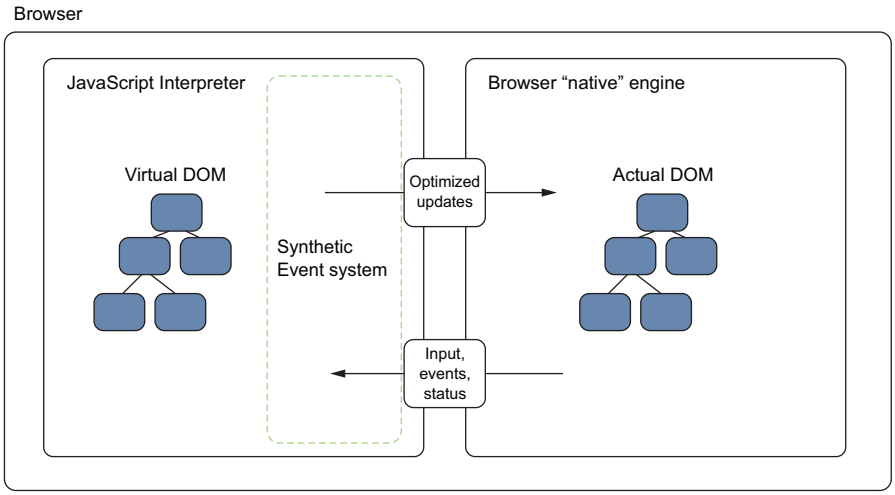

Figure 1.3 The DOM and virtual DOM. React's virtual DOM handles change detection in data as well as translating browser events into events that React components can understand and react to. React's virtual DOM also aims to optimize changes made to the DOM for the sake of performance.

1.3.1 The DOM

The best way to ensure that we understand React's virtual DOM is to start by checking our understanding of the DOM. If you already feel you have a deep understanding of

the DOM, feel free to move ahead. But if not, let's start with an important question: what is the DOM? The DOM, or *Document Object Model*, is a programming interface that allows your JavaScript programs to interact with different types of documents (HTML, XML, and SVG). There are standards-driven specifications for it, which means that a public working group has created a standard set of features it should have and ways it should behave. Although other implementations exist, the DOM is mostly synonymous with web browsers like Chrome, Firefox, and Edge.

The DOM provides a structured way of accessing, storing, and manipulating different parts of a document. At a high level, the DOM is a tree structure that reflects the hierarchy of an XML document. This tree structure is comprised of sub-trees that are in turn made of nodes. You'll probably know these as the `div`s and other elements that make up your web pages and applications.

You've probably used the DOM API before—but you may not have known you were using it. Whenever you use a method in JavaScript that accesses, modifies, or stores information related to something in an HTML document, you're almost certainly using the DOM or its related APIs (see https://developer.mozilla.org/en-US/docs/Web/API for more on web APIs). This means that not all the methods you've used in JavaScript are necessarily part of the JavaScript language itself (`document.findElemenyById`, `querySelectorAll`, `alert`, and so on). They're part of the bigger collection of *web APIs*—the DOM and other APIs that go into a browser—that allow you to interact with documents. Figure 1.4 shows a simplified version of the DOM tree structure you've probably seen in your web pages.

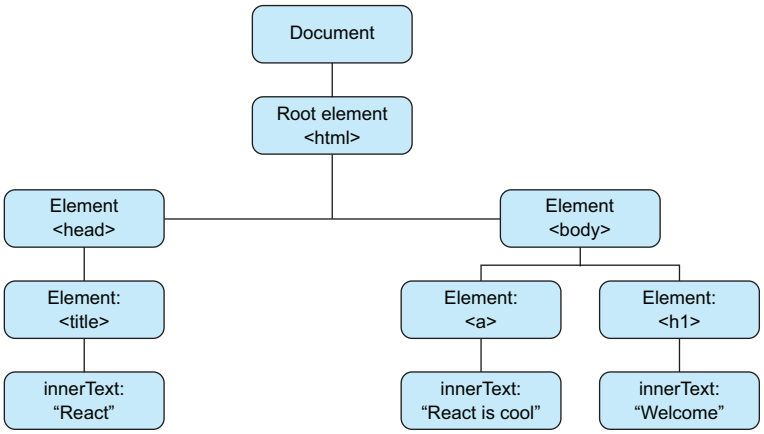

Figure 1.4 Here's a simple version of the DOM tree structure, using elements you're probably familiar with. The DOM API that's exposed to JavaScript lets you performs operations on these elements in the tree.

Common methods or properties you may have used to update or query a web page might include `getElementById`, `parent.appendChild`, `querySelectorAll`, `innerHTML`,

and others. These are all provided by the host environment (in this case, the browser) and allow JavaScript to interact with the DOM. Without this ability, we'd have far less interesting web apps to use and perhaps no books about React to write!

Interacting with the DOM is usually straightforward but can get complicated in the context of a large web application. Fortunately, we don't often need to directly interact with the DOM when building applications with React—we mostly leave that to React. There are cases when we want to reach out past the virtual DOM and interact with the DOM directly, and we'll cover those in future chapters.

1.3.2 The virtual DOM

The web APIs in browsers let us interact with web documents with JavaScript via the DOM. But if we can already do this, why do we need something else in between? I want to first state that React's implementation of a virtual DOM doesn't mean that the regular web APIs are bad or inferior to React. Without them, React can't work. There are, however, certain pain points of working directly with the DOM in larger web applications. Generally, these pain points arise in the area of change detection. When data changes, we want to update the UI to reflect that. Doing that in a way that's efficient and easy to think about can be difficult, so React aims to solve that problem.

Part of the reason for that problem is the way browsers handle interactions with the DOM. When a DOM element is accessed, modified, or created, the browser is often performing a query across a structured tree to find a given element. That's just to access an element, which is usually only the first part of an update. More often than not, it may have to reperform layout, sizing, and other actions as part of a *mutation*—all of which can tend to be computationally expensive. A virtual DOM won't get you around this, but it can help updates to the DOM be optimized to account for these constraints.

When creating and managing a sizeable application that deals with data that changes over time, many changes to the DOM may be required, and often these changes can conflict or are done in a less-than-optimal way. That can result in an overly complicated system that's difficult for engineers to work on and likely a subpar experience for users—lose-lose. Thus performance is another key consideration in React's design and implementation. Implementing a virtual DOM helps address this, but it should be noted that it's designed to be just "fast enough." A robust API, simple mental model, and other things like cross-browser compatibility end up being more important outcomes of React's virtual DOM than an extreme focus on performance. The reason I make this point is that you may hear the virtual DOM talked about as a sort of silver bullet for performance. It is performant, but it's no magic performance bullet, and at the end of the day, many of its other benefits are more important for working with React.

1.3.3 Updates and diffing

How does the virtual DOM work? React's virtual DOM has a few similarities to another software world: 3D gaming. 3D games sometimes employ a rendering process that works very roughly as follows: get information from the game server, send it to the game world (the visual representation that the user sees), determine what changes need to be made to the visual world, and then let the graphics card determine the minimum changes necessary. One advantage of this approach is that you only need the resources for dealing with incremental changes and can generally do things much quicker than if you had to update everything.

That's a gross oversimplification of the way 3D games are rendered and updated, but the general ideas give us a good example to think of when looking at how React performs updates. DOM mutation done poorly can be expensive, so React tries to be efficient in its updates to your UI and employs methods similar to 3D games.

As figure 1.5 shows, React creates and maintains a virtual DOM in memory, and a renderer like React-DOM handles updating the browser DOM based on changes. React can perform intelligent updates and only do work on parts that have changed because it can use *heuristic diffing* to calculate which parts of the in-memory DOM require changes to the DOM. Theoretically, this is much more streamlined and elegant than "dirty checking" or other more brute-force approaches, but a major practical implication is that developers have less complicated state tracking to reason about.

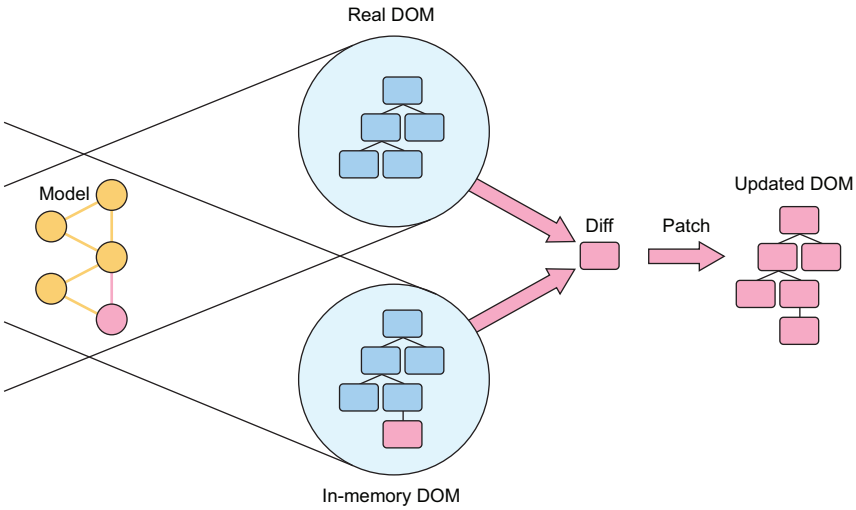

Figure 1.5 React's diffing and update procedure. When a change happens, React determines differences between the actual and in-memory DOMs. Then it performs an efficient update to the browser's DOM. This process is often referred to as a *diff* ("what changed?") *and patch* ("update only what changed") process.

1.3.4 Virtual DOM: Need for speed?

As I've noted, there's more to the virtual DOM than speed. It's performant by design and generally results in snappy, speedy applications that are fast enough for modern web application needs. Performance and a better mental model have been so appreciated by engineers that many popular JavaScript libraries are creating their own versions or variations of a virtual DOM. Even in these cases, people tend to think that the virtual DOM is primarily focused on performance. Performance is a key feature of React, but it's secondary to simplicity. The virtual DOM is part of what enables you to defer thinking about complicated state logic and focus on other, more important parts of your application. Together, speed and simplicity mean happier users and happier developers—a win-win!

I've spent some time talking about the virtual DOM, but I don't want to give you the idea that it will be an important part of working with React. In practice, you won't need to be thinking extensively about how the virtual DOM is accomplishing your data updates or making your changes to your application. That's part of the simplicity of React: you're freed up to focus on the parts of your application that need the most focus.

1.4 Components: The fundamental unit of React

React doesn't just use a novel approach to dealing with changing data over time; it also focuses on components as a paradigm for organizing your application. Components are the most fundamental unit of React. There are several different ways you can create components with React, which future chapters will cover. Thinking in terms of components is essential for grasping not only how React was meant to work but also how you can best use it in your projects.

1.4.1 Components in general

What is a component? It's a part of a larger whole. The idea of components is likely familiar to you, and you probably see them often even though you might not realize it. Using components as mental and visual tools when designing and building user interfaces can lead to better, more intuitive application design and use. A component can be whatever you determine it to be, although not everything makes sense as a component. For example, if you decide that the entirety of an interface is a component, with no child components or further subdivisions, you're probably not helping yourself. Instead, it's helpful to break different parts of an interface into parts that can be composed, reused, and easily reorganized.

To start thinking in terms of components, we'll look at an example interface and break it down into its constituent parts. Figure 1.6 shows an example of an interface you'll be working on later in the book. User interfaces often contain elements that are reused or repurposed in other parts of the interface. And even if they're not reused, they're at least distinct. These different elements, the distinct elements of an interface, can be thought of as components. The interface on the left in figure 1.6 is broken down into components on the right.

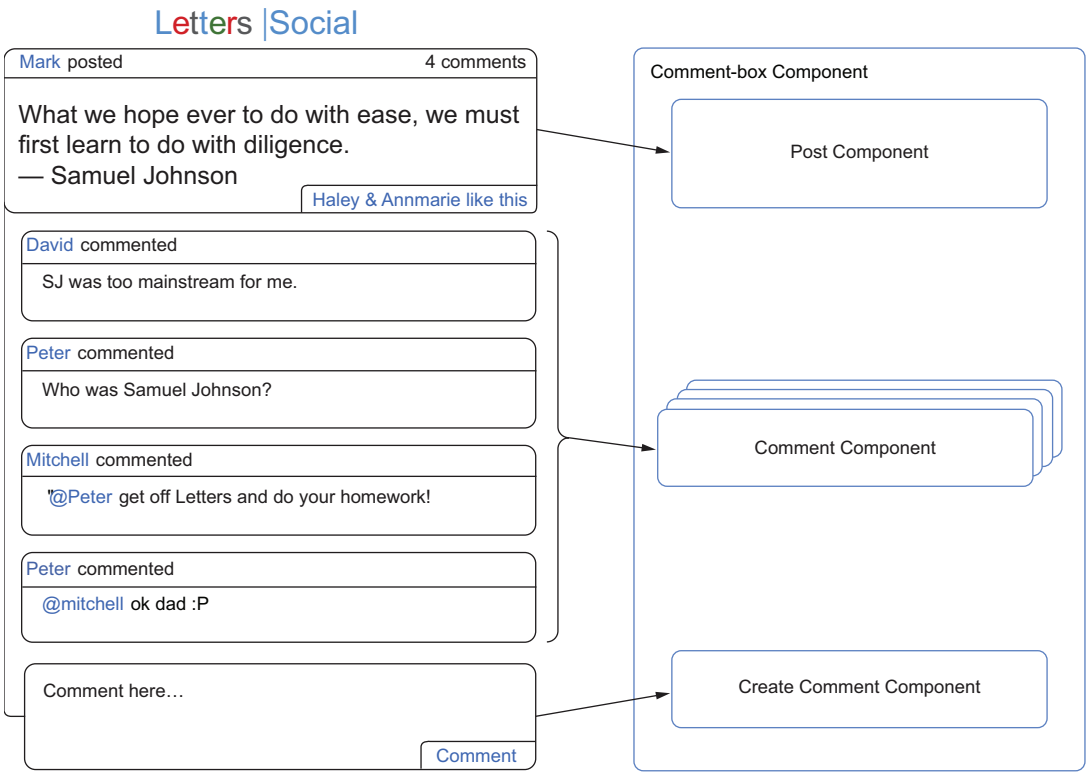

Figure 1.6 An example of an interface broken into components. Each distinct section can be thought of as a component. Items that repeat in a uniform nature can be thought of as one component that gets reused over different data.

> **Exercise 1.1 Component thinking**
>
> Visit a popular site that you enjoy and use often (like GitHub, for example) and break down the interface into components. As you go, you'll probably find yourself dividing things into separate parts. When does it make sense to stop breaking things down? Should an individual letter be a component? When might it make sense for a component to be something small? When would it make sense to consider a grouping of things as one component?

1.4.2 Components in React: Encapsulated and reusable

React components are well encapsulated, reusable, and composable. These characteristics help enable a simpler and more elegant way of thinking about and building user interfaces. Your application can be comprised of clear, concise groups instead of being a spaghetti-code mess. Using React to build your application is almost like

building your project with LEGOs, except that you can't run out of pieces. You'll encounter bugs, but thankfully there are no pieces to step on.

In exercise 1.1, you practiced thinking with components and broke an interface into some constituent components. You could have done it any number of ways, and it's possible you might not have been especially organized or consistent. That's fine. But when you work with components in React, it will be important to consider organization and consistency in component design. You'll want to design components that are self-contained and focus on a particular concern or a handful of related concerns.

This lends itself towards components that are portable, logically grouped, and easy to move around and reuse throughout your application. Even if it takes advantage of other libraries, a well-designed React component should be fairly self-contained. Breaking your UI into components allows you to work more easily on different parts of the application. Boundaries between components mean that functionality and organization can be well-defined, whereas self-contained components mean they can be reused and moved around more easily.

Components in React are meant to work together. This means you can *compose* together components to form new *composite* components. Component composition is one of the most powerful aspects of React. You can create a component once and make it available to the rest of your application for reuse. This is often especially helpful in larger applications. If you're on a medium-to-large team, you could publish components to a private registry (npm or otherwise) that other teams could easily pull down and use in new or existing projects. This might not be a realistic scenario for all sizes of teams, but even smaller teams will benefit from the code reuse that React components promote.

A final aspect of React components is *lifecycle methods*. These are predictable, well-defined methods you can use as your component moves through different parts of its lifecycle (mounting, updating, unmounting, and so on). We'll spend a lot of time on these methods in future chapters.

1.5 React for teams

You now know a little bit more about components in React. React can make your life easier as an individual developer. But what about on a team? Overall, what makes React so appealing to individual developers is also what can make it a great fit for teams. Like any technology, React isn't a perfect solution for every use case or project, no matter the hype or what fanatical developers may try to convince you of. As you've already seen, there are many things that React doesn't do. But the things it does do, it does extremely well.

What makes React a great tool for larger teams and larger applications? First, there's the simplicity of using it. *Simplicity* is not the same thing as *ease*. Easy solutions are often dirty and quick, and worst of all, they can incur technical debt. Truly simple technology is flexible and robust. React provides powerful abstractions that can still

be worked with along with ways to drop down into the lower-level details when necessary. Simple technology is easier to understand and work with because the difficult work of streamlining and removing what's not necessary has been done. In many ways React has made simple easy, providing an effective solution without introducing harmful "black magic" or an opaque API.

All this is great for the individual developer, but the effect is amplified across larger teams and organizations. Although there's certainly room for React to improve and keep growing, the hard work of making it a simple and flexible technology pays off for engineering teams. Simpler technologies with good mental models tend to create less of a mental burden for engineers and let them move faster and have a higher impact. As a bonus, a simpler set of tools is easier to learn for new employees. Trying to ramp up a new team member to an overly complex stack will not only cost time for the training engineers, it will also probably mean that the new developer will be unable to make meaningful contributions for some time. Because React seeks to carefully rethink established best practices, there's the initial cost in paradigm switch, but after that it's often a big, long-term win.

Although it's certainly a different tool than others in the same space, React is a fairly lightweight library in terms of responsibility and functionality. Where something like Angular may require you to "buy in" to a more comprehensive API, React is only concerned with the view of your application. This means it's much more trivial to integrate it with your current technologies, and it will leave you room to make choices about other aspects. Some opinionated frameworks and libraries require an all-or-nothing adoption stance, but React's "just the view" scope and general interoperability with JavaScript mean this isn't always the case.

Instead of going all-in, you can incrementally transition different projects or tools over to React without having to make a drastic change to your structure, build stack, or other related areas. That's a desirable trait for almost any technology, and it's how React was first tried out at Facebook—in one small project area. From there it grew and took hold as more and more teams saw and experienced its benefits. What does all this mean for your team? It means you can evaluate React without having to take the risk of completely rewriting the product using React.

The simplicity, un-opinionated nature, and performance of React make it a great fit for projects small and large alike. As you keep exploring React, you'll see how it can be a good fit for your team and projects.

1.6 Summary

React is a library for creating user interfaces that was initially built and open sourced by Facebook. It's a JavaScript library built with simplicity, performance, and components in mind. Rather than provide a comprehensive set of tools for creating applications, it allows you to choose how to implement your data models, server calls, and other application concerns, and what to implement them with. These key reasons and others are why React can be a great tool for small and large applications

and teams alike. Here are some of the benefits of React briefly summarized for a few typical roles:

- *Individual developer*—Once you learn React, your applications can be easier to rapidly build out. They will tend to be easier to work on for larger teams, and sophisticated features can be easier to implement and maintain.
- *Engineering manager*—There's an initial cost for developers as they learn React, but eventually they'll be able to more easily and quickly develop complex applications.
- *CTO or upper management*—React, like any technology, is an investment with risks. But the eventual gains in productivity and reduced mental burdens often outweigh time sunk into ramping up. That's not the case for every team, but it's true for many.

All in all, React can be relatively easy for onboarding engineers to learn, can reduce the total amount of unnecessary complexity in an application, and can reduce technical debt by promoting code reuse. Take a second to review some of what you've learned about React so far:

- React is a library for building user interfaces, originally created by engineers at Facebook.
- React provides a simple, flexible API that's based around components.
- Components are the fundamental unit of React, and they're used extensively in React applications.
- React implements a virtual DOM that sits between your program and the browser DOM.
- The virtual DOM allows for efficient updates to the DOM using a fast diffing algorithm.
- The virtual DOM allows for excellent performance, but the biggest win is the mental model that it affords.

Now that you know a little more about the background and design of React, we can really dive in. In the next chapter, you'll create your first component and take a closer look at how React works. You'll be learning more about the virtual DOM, components in React, and how you can create components of your own.

JAVASCRIPT/WEB DEVELOPMENT

React IN ACTION

Mark Tielens Thomas

Facebook created React to help deliver amazing user experiences on a website with thousands of components and an incomprehensible amount of traffic. The same powerful tools are available to you too! The key is a clever design for managing state, data flow, and rendering, so your application is easy to think about and runs smoothly. Add an incredibly rich ecosystem of components and libraries, and you've got a recipe for building web apps that will delight both developers and users.

React in Action teaches you to think like a pro about user interfaces and building them with React. This practical book gets you up and running quickly with hands-on examples in every chapter. You'll master core topics like rendering, lifecycle methods, JSX, data flow, forms, routing, integrating with third-party libraries, and testing. And the included application design ideas will help make your apps pop. As you learn to integrate React into full-stack applications, you'll explore state management with Redux and server-side rendering, and even dabble in React Native for mobile UIs.

What's Inside

- React from the ground up
- Implementing a routing system with components
- Server-side rendering in Node.js
- Working with third-party libraries
- Testing React components

Written for developers familiar with HTML, CSS, and JavaScript.

Mark Thomas is an experienced software engineer who works daily with React, JavaScript, and Node.js. He loves clean code, beautiful systems, and good coffee.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/react-in-action

"Read this. Work with React. Never look back."
—Michal Paszkiewicz
Transport for London

"One stop—for concepts as well as for real-world examples and integrations."
—Phaneendra Bommareddy
Openlogix

"A must-have for anyone wanting to create applications using React and Redux!"
—Andrew Courter, Pivotal

"Easy to follow, clearly demonstrates all necessary steps, includes plenty of code examples, and never leaves you in the dark."
—Olivier Ducatteeuw
University of Leuven

ISBN-13: 978-1-61729-385-6
ISBN-10: 1-61729-385-7
54499
9 781617 293856

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]