Watch this chapter's introductory video by scanning this QR code with your phone or going to http://reactquickly.co/videos/ch01.

1 Meeting React

This chapter covers

- Understanding what React is
- Solving problems with React
- Fitting React into your web applications
- Writing your first React app: Hello World

When I began working on web development in early 2000, all I needed was some HTML and a server-side language like Perl or PHP. Ah, the good old days of putting in `alert()` boxes just to debug your front-end code. It's a fact that as the internet has evolved, the complexity of building websites has increased dramatically. Websites have become web applications with complex user interfaces, business logic, and data layers that require changes and updates over time—and often in real time.

Many JavaScript template libraries have been written to try to solve the problems of dealing with complex user interfaces (UIs). But they still require developers to adhere to the old separation of concerns—which splits style (CSS), data and structure (HTML), and dynamic interactions (JavaScript)—and they don't meet modern-day needs. (Remember the term *DHTML*?)

In contrast, React offers a new approach that streamlines front-end development. React is a powerful UI library that offers an alternative that many big firms such as Facebook, Netflix, and Airbnb have adopted and see as the way forward. Instead of defining a one-off template for your UIs, React allows you to create reusable UI components in JavaScript that you can use again and again in your sites.

Do you need a captcha control or date picker? Then use React to define a `<Captcha />` or `<DatePicker />` component that you can add to your form: a simple drop-in component with all the functionality and logic to communicate with the back end. Do you need an autocomplete box that asynchronously queries a database once the user has typed four or more letters? Define an `<Autocomplete charNum="4"/>` component to make that asynchronous query. You can choose whether it has a text box UI or has no UI and instead uses another custom form element—perhaps `<Autocomplete textbox="..." />`.

This approach isn't new. Creating *composable UIs* has been around for a long time, but React is the first to use pure JavaScript without templates to make this possible. And this approach has proven easier to maintain, reuse, and extend.

React is a great library for UIs, and it should be part of your front-end web toolkit; but it isn't a complete solution for all front-end web development. In this chapter, we'll look at the pros and cons of using React in your applications and how you might fit it into your existing web-development stack.

Part 1 of the book focuses on React's primary concepts and features, and part 2 looks at working with libraries related to React to build more-complex front-end apps (a.k.a. *React stack* or *React and friends*). Each part demonstrates both greenfield and brownfield development[1] with React and the most popular libraries, so you can get an idea of how to approach working with it in real-world scenarios.

Chapter videos and source code

We all learn differently. Some people prefer text and others video, and others learn best via in-person instruction. Each chapter of this book includes a short video that explains the chapter's gist in less than 5 minutes. Watching them is totally *optional*. They'll give you a summary if you prefer a video format or need a refresher. After watching each video, you can decide whether you need to read the chapter or can skip to the next one.

The source code for the examples in this chapter is at www.manning.com/books/react-quickly and at https://github.com/azat-co/react-quickly/tree/master/ch01 (in the ch01 folder of the GitHub repository https://github.com/azat-co/react-quickly). You can also find some demos at http://reactquickly.co/demos.

[1] *Brownfield* is a project with legacy code and existing systems, while *greenfield* is a project without any legacy code or systems; see https://en.wikipedia.org/wiki/Brownfield_(software_development).

1.1 What is React?

To introduce React.js properly, I first need to define it. So, what is React? It's a UI component library. The UI components are created with React using JavaScript, not a special template language. This approach is called *creating composable UIs*, and it's fundamental to React's philosophy.

React UI components are highly self-contained, concern-specific blocks of functionality. For example, there could be components for date-picker, captcha, address, and ZIP code elements. Such components have both a visual representation and dynamic logic. Some components can even talk to the server on their own: for example, an autocomplete component might fetch the autocompletion list from the server.

User interfaces

In a broad sense, a user interface[2] is everything that facilitates communication between computers and humans. Think of a punch card or a mouse: they're both UIs. When it comes to software, engineers talk about graphical user interfaces (GUIs), which were pioneered for early personal computers such as Macs and PCs. A GUI consists of menus, text, icons, pictures, borders, and other elements. Web elements are a narrow subset of the GUI: they reside in browsers, but there are also elements for desktop applications in Windows, OS X, and other operating systems.

Every time I mention a *UI* in this book, I mean a *web GUI*.

Component-based architecture (CBA)—not to be confused with web components, which are just one of the most recent implementations of CBA—existed before React. Such architectures generally tend to be easier to reuse, maintain, and extend than monolithic UIs. What React brings to the table is the use of pure JavaScript (without templates) and a new way to look at composing components.

1.2 The problem that React solves

What problem does React solve? Looking at the last few years of web development, note the problems in building and managing complex web UIs for front-end applications: React was born primarily to address those. Think of large web apps like Facebook: one of the most painful tasks when developing such applications is managing how the views change in response to data changes.

Let's refer to the official React website for more hints about the problem React addresses: "We built React to solve one problem: building large applications with data that changes over time."[3] Interesting! We can also look at the history of React for more information. A discussion on the React Podcast[4] mentions that the creator of

---

[2] https://en.wikipedia.org/wiki/User_interface.

[3] React official website, "Why React?" March 24, 2016, http://bit.ly/2mdCJKM.

[4] *React Podcast*, "8. React, GraphQL, Immutable & Bow-Ties with Special Guest Lee Byron," December 31, 2015, http://mng.bz/W1X6.

React—Jordan Walke—was solving a problem at Facebook: having multiple data sources update an autocomplete field. The data came asynchronously from a back end. It was becoming more and more complicated to determine where to insert new rows in order to reuse DOM elements. Walke decided to generate the field representation (DOM elements) anew each time. This solution was elegant in its simplicity: UIs as functions. Call them with data, and you get rendered views predictably.

Later, it turned out that generating elements in memory is extremely fast and that the actual bottleneck is rendering in the DOM. But the React team came up with an algorithm that avoids unnecessary DOM pain. This made React very fast (and cheap in terms of performance). React's splendid performance and developer-friendly, component-based architecture are a winning combination. These and other benefits of React are described in the next section.

React solved Facebook's original problem, and many large firms agreed with this approach. React adoption is solid, and its popularity is growing every month. React emerged from Facebook[5] and is now used not only by Facebook but also by Instagram, PayPal, Uber, Sberbank, Asana,[6] Khan Academy,[7] HipChat,[8] Flipboard,[9] and Atom,[10] to name just a few.[11] Most of these applications originally used something else (typically, template engines with Angular or Backbone) but switched to React and are extremely happy about it.

1.3 Benefits of using React

Every new library or framework claims to be better than its predecessors in some respect. In the beginning, we had jQuery, and it was leaps and bounds better for writing cross-browser code in native JavaScript. If you remember, a single AJAX call taking many lines of code had to account for Internet Explorer and WebKit-like browsers. With jQuery, this takes only a single call: `$.ajax()`, for example. Back in the day, jQuery was called a framework—but not anymore! Now a *framework* is something bigger and more powerful.

Similarly with Backbone and then Angular, each new generation of JavaScript frameworks has brought something new to the table. React isn't unique in this. What *is* new is that React challenges some of the core concepts used by most popular front-end frameworks: for example, the idea that you need to have templates.

The following list highlights some of the benefits of React versus other libraries and frameworks:

---

[5] "Introduction to React.js," July 8, 2013, http://mng.bz/86XF.

[6] Malcolm Handley and Phips Peter, "Why Asana Is Switching to TypeScript," *Asana Blog*, November 14, 2014, http://mng.bz/zXKo.

[7] Joel Burget, "Backbone to React," http://mng.bz/WGEQ.

[8] Rich Manalang, "Rebuilding HipChat with React.js," *Atlassian Developers*, February 10, 2015, http://mng.bz/r0w6.

[9] Michael Johnston, "60 FPS on the Mobile Web," *Flipboard*, February 10, 2015, http://mng.bz/N5F0.

[10] Nathan Sobo, "Moving Atom to React," *Atom*, July 2, 2014, http://mng.bz/K94N.

[11] See also the JavaScript usage stats at http://libscore.com/#React.

- *Simpler apps*—React has a CBA with pure JavaScript; a declarative style; and powerful, developer-friendly DOM abstractions (and not just DOM, but also iOS, Android, and so on).
- *Fast UIs*—React provides outstanding performance thanks to its virtual DOM and smart-reconciliation algorithm, which, as a side benefit, lets you perform testing without spinning up (starting) a headless browser.
- *Less code to write*—React's great community and vast ecosystem of components provide developers with a variety of libraries and components. This is important when you're considering what framework to use for development.

Many features make React simpler to work with than most other front-end frameworks. Let's unpack these items one by one, starting with its simplicity.

1.3.1 Simplicity

The concept of simplicity in computer science is highly valued by developers and users. It doesn't equate to ease of use. Something simple can be hard to implement, but in the end it will be more elegant and efficient. And often, an easy thing will end up being complex. Simplicity is closely related to the KISS principle (keep it simple, stupid).[12] The gist is that simpler systems work better.

React's approach allows for simpler solutions via a dramatically better web-development experience for software engineers. When I began working with React, it was a considerable shift in a positive direction that reminded me of switching from using plain, no-framework JavaScript to jQuery.

In React, this simplicity is achieved with the following features:

- *Declarative over imperative style*—React embraces declarative style over imperative by updating views automatically.
- *Component-based architecture using pure JavaScript*—React doesn't use domain-specific languages (DSLs) for its components, just pure JavaScript. And there's no separation when working on the same functionality.
- *Powerful abstractions*—React has a simplified way of interacting with the DOM, allowing you to normalize event handling and other interfaces that work similarly across browsers.

Let's cover these one by one.

Declarative over imperative style

First, React embraces declarative style over imperative. Declarative style means developers write how it *should* be, not what to do, step-by-step (imperative). But why is declarative style a better choice? The benefit is that declarative style reduces complexity and makes your code easier to read and understand.

Consider this short JavaScript example, which illustrates the difference between declarative and imperative programming. Let's say you need to create an array (`arr2`)

[12] https://en.wikipedia.org/wiki/KISS_principle.

whose elements are the result of doubling the elements of another array (`arr`). You can use a `for` loop to iterate over an array and tell the system to multiply by 2 and create a new element (`arr2[i]=`):

```
var arr = [1, 2, 3, 4, 5],
  arr2 = []
for (var i=0; i<arr.length; i++) {
  arr2[i] = arr[i]*2
}
console.log('a', arr2)
```

The result of this snippet, where each element is multiplied by 2, is printed on the console as follows:

```
a [2, 4, 6, 8, 10]
```

This illustrates imperative programming, and it works—until it doesn't work, due to the complexity of the code. It becomes too difficult to understand what the end result is supposed to be when you have too many imperative statements. Fortunately, you can rewrite the same logic in declarative style with `map()`:

```
var arr = [1, 2, 3, 4, 5],
  arr2 = arr.map(function(v, i){ return v*2 })
console.log('b', arr2)
```

The output is `b [2, 4, 6, 8, 10]`; the variable `arr2` is the same as in the previous example. Which code snippet is easier to read and understand? In my humble opinion, the declarative example.

Look at the following imperative code for getting a nested value of an object. The expression needs to return a value based on a string such as `account` or `account.number` in such a manner that these statements print `true`:

```
var profile = {account: '47574416'}
var profileDeep = {account: { number: 47574416 }}
console.log(getNestedValueImperatively(profile, 'account') === '47574416')
console.log(getNestedValueImperatively(profileDeep, 'account.number')
➥ === 47574416)
```

This imperative style literally tells the system what to do to get the results you need:

```
var getNestedValueImperatively = function getNestedValueImperatively
➥ (object, propertyName) {
  var currentObject = object
  var propertyNamesList = propertyName.split('.')
  var maxNestedLevel = propertyNamesList.length
  var currentNestedLevel

  for (currentNestedLevel = 0; currentNestedLevel < maxNestedLevel;
  ➥ currentNestedLevel++) {
```

```
    if (!currentObject || typeof currentObject === 'undefined')
    ➥ return undefined
    currentObject = currentObject[propertyNamesList[currentNestedLevel]]
  }

  return currentObject
}
```

Contrast this with declarative style (focused on the result), which reduces the number of local variables and thus simplifies the logic:

```
var getValue = function getValue(object, propertyName) {
  return typeof object === 'undefined' ? undefined : object[propertyName]
}

var getNestedValueDeclaratively = function getNestedValueDeclaratively(object,
➥ propertyName) {
  return propertyName.split('.').reduce(getValue, object)
}
console.log(getNestedValueDeclaratively({bar: 'baz'}, 'bar') === 'baz')
console.log(getNestedValueDeclaratively({bar: { baz: 1 }}, 'bar.baz')=== 1)
```

Most programmers have been trained to code imperatively, but usually the declarative code is simpler. In this example, having fewer variables and statements makes the declarative code easier to grasp at first glance.

That was just some JavaScript code. What about React? It takes the same declarative approach when you compose UIs. First, React developers describe UI elements in a declarative style. Then, when there are changes to views generated by those UI elements, React takes care of the updates. Yay!

The convenience of React's declarative style fully shines when you need to make changes to the view. Those are called changes of the *internal state*. When the state changes, React updates the view accordingly.

NOTE I cover how states work in chapter 4.

Under the hood, React uses a *virtual DOM* to find differences (the delta) between what's already in the browser and the new view. This process is called *DOM diffing* or *reconciliation of state and view* (bringing them back to similarity). This means developers don't need to worry about explicitly changing the view; all they need to do is update the state, and the view will be updated automatically as needed.

Conversely, with jQuery, you'd need to implement updates imperatively. By manipulating the DOM, developers can programmatically modify the web page or parts of the web page (a more likely scenario) without rerendering the entire page. DOM manipulation is what you do when you invoke jQuery methods.

Some frameworks, such as Angular, can perform automatic view updates. In Angular, it's called *two-way data binding*, which basically means views and models have two-way communication/syncing of data between them.

The jQuery and Angular approaches aren't great, for two reasons. Think about them as two extremes. At one extreme, the library (jQuery) isn't doing anything, and a developer (you!) needs to implement all the updates manually. At the other extreme, the framework (Angular) is doing everything.

The jQuery approach is prone to mistakes and takes more work to implement. Also, this approach of directly manipulating the regular DOM works fine with simple UIs, but it's limiting when you're dealing with a lot of elements in the DOM tree. This is the case because it's harder to see the results of imperative functions than declarative statements.

The Angular approach is difficult to reason about because with its two-way binding, things can spiral out of control quickly. You insert more and more logic, and all of a sudden, different views are updating models, and those models update other views.

Yes, the Angular approach is somewhat more readable than imperative jQuery (and requires less manual coding!), but there's another issue. Angular relies on templates and a DSL that uses `ng` directives (for example, `ng-if`). I discuss its drawbacks in the next section.

COMPONENT-BASED ARCHITECTURE USING PURE JAVASCRIPT

Component-based architecture[13] existed before React came on the scene. Separation of concerns, loose coupling, and code reuse are at the heart of this approach because it provides many benefits; software engineers, including web developers, love CBA. A building block of CBA in React is the component class. As with other CBAs, it has many benefits, with code reuse being the main one (you can write less code!).

What was lacking before React was a pure JavaScript implementation of this architecture. When you're working with Angular, Backbone, Ember, or most of the other MVC-like front-end frameworks, you have one file for JavaScript and another for the template. (Angular uses the term *directives* for components.) There are a few issues with having two languages (and two or more files) for a single component.

The HTML and JavaScript separation worked well when you had to render HTML on the server, and JavaScript was only used to make your text blink. Now, single page applications (SPAs) handle complex user input and perform rendering on the browser. This means HTML and JavaScript are closely coupled functionally. For developers, it makes more sense if they don't need to separate between HTML and JavaScript when working on a piece of a project (component).

Consider this Angular code, which displays different links based on the value of `userSession`:

```
<a ng-if="user.session" href="/logout">Logout</a>
<a ng-if="!user.session" href="/login">Login</a>
```

You can read it, but you may have doubts about what `ng-if` takes: a Boolean or a string. And will it hide the element or not render it at all? In the Angular case, you can't be sure whether the element will be hidden on true or false, unless you're familiar with how this particular `ng-if` directive works.

[13] http://mng.bz/a65r.

Compare the previous snippet with the following React code, which uses JavaScript `if/else` to implement conditional rendering. It's absolutely clear what the value of `user.session` must be and what element (`logout` or `login`) is rendered if the value is true. Why? Because it's just JavaScript:

```
if (user.session) return React.createElement('a', {href: '/logout'}, 'Logout')
else return React.createElement('a', {href: '/login'}, 'Login')
```

Templates are useful when you need to iterate over an array of data and print a property. We work with lists of data all the time! Let's look at a `for` loop in Angular. As mentioned earlier, in Angular, you need to use a DSL with directives. The directive for a `for` loop is `ng-repeat`:

```
<div ng-repeat="account in accounts">
  {{account.name}}
</div>
```

One of the problems with templates is that developers often have to learn yet another language. In React, you use pure JavaScript, which means you don't need to learn a new language! Here's an example of composing a UI for a list of account names with pure JavaScript:

```
accounts.map(function(account)  {
  return React.createElement('div', null, account.name)
})
```

Regular JavaScript method that takes an iterator expression as a parameter[14]

Iterator expression that returns a with the account name

Imagine a situation where you're making some changes to the list of accounts. You need to display the account number and other fields. How do you know what fields the account has in addition to `name`?

You need to open the corresponding JavaScript file that calls and uses this template, and then you have to find `accounts` to see its properties. So the second problem with templates is that the logic about the data and the description of how that data should be rendered are separated.

It's much better to have the JavaScript and the markup in one place so you don't have to switch between file and languages. This is exactly how React works; and you'll see how React renders elements shortly in a Hello World example.

> **NOTE** Separation of concerns generally is a good pattern. In a nutshell, it means separation of different functions such as the data service, the view layer, and so on. When you're working with template markup and corresponding JavaScript code, you're working on *one functionality*. That's why having two files (.js and .html) isn't a separation of concerns.

[14] http://mng.bz/555J.

Now, if you want to explicitly set the method by which to keep track of items (for example, to ensure there are no duplicates) in the rendered list, you can use Angular's `track by` feature:

```
<div ng-repeat="account in accounts track by account._id">
  {{account.name}}
</div>
```

If you want to track by an index of the array, there's `$index`:

```
<div ng-repeat="account in accounts track by $index">
  {{account.name}}
</div>
```

But what concerns me and many other developers is, what is this magic `$index`? In React, you use an argument from `map()` for the value of the `key` attribute:

```
accounts.map(function(account, index)  {
  return React.createElement('div', {key: index}, account.name)
})
```

Uses an array element value (account) and its index provided by Array.map()

Returns a React element with an attribute key with the value index and inner text set to account.name

It's worth noting that `map()` isn't exclusive to React. You can use it with other frameworks because it's part of the language. But the declarative nature of `map()` makes it and React a perfect pair.

I'm not picking on Angular—it's a great framework. But the bottom line is that if a framework uses a DSL, you need to learn its magic variables and methods. In React, you can use pure JavaScript.

If you use React, you can carry your knowledge to the next project even if it's not in React. On the other hand, if you use an X template engine (or a Y framework with a built-in DSL template engine), you're locked into that system and have to describe yourself as an X/Y developer. Your knowledge isn't transferable to projects that don't use X/Y. To summarize, the pure JavaScript component-based architecture is about using discrete, well-encapsulated, reusable components that ensure better separation of concerns based on functionality without the need for DSLs, templates, or directives.

Working with many developer teams, I've observed another factor related to simplicity. React has a better, shallower, more gradual learning curve compared to MVC frameworks (well, React isn't an MVC, so I'll stop comparing them) and template engines that have special syntax—for example, Angular directives or Jade/Pug. The reason is that instead of using the power of JavaScript, most template engines build abstractions with their own DSL, in a way reinventing things like an `if` condition or a `for` loop.

POWERFUL ABSTRACTIONS

React has a powerful abstraction of the document model. In other words, it hides the underlying interfaces and provides normalized/synthesized methods and properties. For example, when you create an `onClick` event in React, the event handler will receive not a native browser-specific event object, but a synthetic event object that's a wrapper around native event objects. You can expect the same behavior from synthetic events regardless of the browser in which you run the code. React also has a set of synthetic events for touch events, which are great for building web apps for mobile devices.

Another example of React's DOM abstraction is that you can render React elements on the server. This can be handy for better search engine optimization (SEO) and/or improving performance.

There are more options when it comes to rendering React components than just DOM or HTML strings for the server back end. We'll cover them in section 1.5.1. And, speaking of the DOM, one of the most sought-after benefits of React is its splendid performance.

1.3.2 Speed and testability

In addition to the necessary DOM updates, your framework may perform unnecessary updates, which makes the performance of complex UIs even worse. This becomes especially noticeable and painful for users when you have a lot of dynamic UI elements on your web page.

On the other hand, React's virtual DOM exists only in the JavaScript memory. Every time there's a data change, React first compares the differences using its virtual DOM; only when the library knows there has been a change in the rendering will it update the actual DOM. Figure 1.1 shows a high-level overview of how React's virtual DOM works when there are data changes.

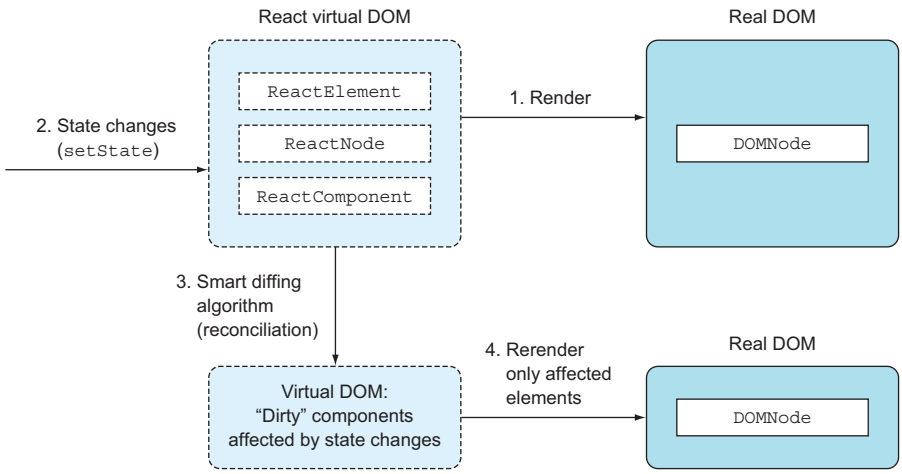

Figure 1.1 Once a component has been rendered, if its state changes, it's compared to the in-memory virtual DOM and rerendered if necessary.

Ultimately, React updates only those parts that are absolutely necessary so that the internal state (virtual DOM) and the view (real DOM) are the same. For example, if there's a `<p>` element and you augment the text via the state of the component, only the text will be updated (that is, `innerHTML`), not the element itself. This results in increased performance compared to rerendering entire sets of elements or, even more so, entire pages (server-side rendering).

> **NOTE** If you like to geek out on algorithms and Big Os, these two articles do a great job of explaining how the React team managed to turn an O(n3) problem into an O(n) one: "Reconciliation," on the React website (http://mng.bz/PQ9X) and "React's Diff Algorithm" by Christopher Chedeau (http://mng.bz/68L4).

The added benefit of the virtual DOM is that you can do unit testing without headless browsers like PhantomJS (http://phantomjs.org). There's a Jasmine (http://jasmine.github.io) layer called Jest (https://facebook.github.io/jest) that lets you test React components right on the command line!

1.3.3 Ecosystem and community

Last, but not least, React is supported by the developers of a juggernaut web application called Facebook, as well as by their peers at Instagram. As with Angular and some other libraries, having a big company behind the technology provides a sound testing ground (it's deployed to millions of browsers), reassurance about the future, and an increase in contribution velocity.

The React community is incredible. Most of the time, developers don't even have to implement much of the code. Look at these community resources:

- List of React components: https://github.com/brillout/awesome-react-components and http://devarchy.com/react-components
- Set of React components that implement the Google Material Design specification (https://design.google.com): http://react-toolbox.com
- Material Design React components: www.material-ui.com
- Collection of React components for Office and Office 360 experiences (http://dev.office.com/fabric#/components) using the Office Design Language: https://github.com/OfficeDev/office-ui-fabric-react
- Opinionated catalog of open source JS (mostly React) packages: https://js.coach
- Catalog of React components: https://react.rocks
- Khan Academy React components: https://khan.github.io/react-components
- Registry of React components: www.reactjsx.com

My personal anecdotal experience with open source taught me that the marketing of open source projects is as important to its wide adoption and success as the code itself. By that, I mean that if a project has a poor website, lacks documentation and examples,

and has an ugly logo, most developers won't take it seriously—especially now, when there are so many JavaScript libraries. Developers are picky, and they won't use an ugly duckling library.

My teacher used to say, "Don't judge a book by its cover." This might sound controversial; but, sadly, most people, including software engineers, are prone to biases such as good branding. Luckily, React has a great engineering reputation backing it. And, speaking of book covers, I hope you didn't buy this book just for its cover!

1.4 Disadvantages of React

Of course, almost everything has its drawbacks. This is true with React, but the full list of cons depends on whom you ask. Some of the differences, like declarative versus imperative, are highly subjective. So, they can be both pros and cons. Here's my list of React disadvantages (as with any such list, it may be biased because it's based on opinions I've heard from other developers):

- React isn't a full-blown, Swiss Army knife–type of framework. Developers need to pair it with a library like Redux or React Router to achieve functionality comparable to Angular or Ember. This can also be an advantage if you need a minimalistic UI library to integrate with your existing stack.
- React isn't as mature as other frameworks. React's core API is still changing, albeit very little after the 0.14 release; the best practices for React (as well as the ecosystem of components, plug-ins, and add-ons) are still developing.
- React uses a somewhat new approach to web development, and JSX and Flux (often used with React as the data library) can be intimidating to beginners. There's a lack of best practices, good books, courses, and resources available for mastering React.
- React only has a one-way binding. Although one-way binding is better for complex apps and removes a lot of complexity, some developers (especially Angular developers) who got used to a two-way binding will find themselves writing a bit more code. I'll explain how React's one-way binding works compared to Angular's two-way binding in chapter 14, which covers working with data.
- React isn't reactive (as in reactive programming and architecture, which are more event-driven, resilient, and responsive) out of the box. Developers need to use other tools such as Reactive Extensions (RxJS, https://github.com/Reactive-Extensions/RxJS) to compose asynchronous data streams with Observables.

To continue with this introduction to React, let's look at how it fits into a web application.

1.5 How React can fit into your web applications

In a way, the React library by itself, without React Router or a data library, is less comparable to frameworks (like Backbone, Ember, and Angular) and more comparable to libraries for working with UIs, like template engines (Handlebars, Blaze) and DOM-manipulation libraries (jQuery, Zepto). In fact, many teams have swapped traditional

template engines like Underscore in Backbone or Blaze in Meteor for React, with great success. For example, PayPal switched from Dust to Angular, as did many other companies listed earlier in this chapter.

You can use React for just part of your UI. For example, let's say you have a load-application form on a web page built with jQuery. You can gradually begin to convert this front-end app to React by first converting the city and state fields to populate automatically based on the ZIP code. The rest of the form can keep using jQuery. Then, if you want to proceed, you can convert the rest of the form elements from jQuery to React, until your entire page is built on React. Taking a similar approach, many teams successfully integrated React with Backbone, Angular, or other existing front-end frameworks.

React is back-end agnostic for the purposes of front-end development. In other words, you don't have to rely on a Node.js back end or MERN (MongoDB, Express.js, React.js, and Node.js) to use React. It's fine to use React with any other back-end technology like Java, Ruby, Go, or Python. React is a UI library, after all. You can integrate it with any back end and any front-end data library (Backbone, Angular, Meteor, and so on).

To summarize how React fits into a web app, it's most often used in these scenarios:

- As a UI library in React-related stack SPAs, such as React+React and Router+Redux
- As a UI library (*V* in MVC) in non-fully React-related stack SPAs, such as React+Backbone
- As a drop-in UI component in *any* front-end stack, such as a React autocomplete input component in a jQuery+server-side rendering stack
- As a server-side template library in a purely thick-server (traditional) web app or in a hybrid or isomorphic/universal web app, such as an Express server that uses `express-react-views`
- As a UI library in mobile apps, such as a React Native iOS app
- As a UI description library for different rendering targets (discussed in the next section)

React works nicely with other front-end technologies, but it's mostly used as part of single-page architecture because SPA seems to be the most advantageous and popular approach to building web apps. I cover how React fits into an SPA in section 1.5.2.

In some extreme scenarios, you can even use React *only on the server* as a template engine of sorts. For example, there's an `express-react-views` library (https://github.com/reactjs/express-react-views). It renders the view server-side from React components. This server-side rendering is possible because React lets you use different rendering targets.

1.5.1 React libraries and rendering targets

In versions 0.14 and higher, the React team split the library into two packages: React Core (`react` package on npm) and ReactDOM (`react-dom` package on npm). By

doing so, the maintainers of React made it clear that React is on a path to become not just a library for the web, but a universal (sometimes called *isomorphic* because it can be used in different environments) library for describing UIs.

For example, in version 0.13, React had a `React.render()` method to mount an element to a web page's DOM node. In versions 0.14 and higher, you need to include `react-dom` and call `ReactDOM.render()` instead of `React.render()`.

Having multiple packages created by the community to support various rendering targets made this approach of separating writing components and rendering logical. Some of these modules are as follows:

- Renderer for the blessed (https://github.com/chjj/blessed) terminal interface: http://github.com/Yomguithereal/react-blessed
- Renderer for the ART library (https://github.com/sebmarkbage/art): https://github.com/reactjs/react-art
- Renderer for `<canvas>`: https://github.com/Flipboard/react-canvas
- Renderer for the 3D library using three.js (http://threejs.org): https://github.com/Izzimach/react-three
- Renderer for virtual reality and interactive 360 experiences: https://facebook.github.io/react-vr

In addition to the support of these libraries, the separation of React Core from ReactDOM makes it easier to share code between React and React Native libraries (used for native mobile iOS and Android development). In essence, when using React for web development, you'll need to include at least React Core and ReactDOM.

Moreover, there are additional React utility libraries in React and npm. (Before React v15.5, some of them were part of React as React *add-ons*.[15] These utility libraries allow you to enhance functionality, work with immutable data (https://github.com/kolodny/immutability-helper), and perform testing.

Finally, React is almost always used with JSX—a tiny language that lets developers write React UIs more eloquently. You can transpile JSX into regular JavaScript by using Babel or a similar tool.

As you can see, there's a lot of modularity—the functionality of React-related things is split into different packages. This gives you power and choice, which is a good thing. No monolith or opinionated library dictates to you the only possible way to implement things. More on this in section 1.5.3.

If you're a web developer reading this book, you probably use SPA architecture. Either you already have a web app built using this and want to reengineer it with React (brownfield), or you're starting a new project from scratch (greenfield). Next, we'll zoom in on React's place in SPAs as the most popular approach to building web apps.

[15] See the version 15.5 change log with the list of add-ons and npm libraries: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html. See also the page on add-ons: https://facebook.github.io/react/docs/addons.html.

1.5.2 Single-page applications and React

Another name for SPA architecture is *thick client*, because the browser, being a client, holds more logic and performs functions such as rendering of the HTML, validation, UI changes, and so on. Figure 1.2 is basic: it shows a bird's-eye view of a typical SPA architecture with a user, a browser, and a server. The figure depicts a user making a request, and input actions like clicking a button, drag-and-drop, mouse hovering, and so on:

1. The user types a URL in the browser to open a new page.
2. The browser sends a URL request to the server.
3. The server responds with static assets such as HTML, CSS, and JavaScript. In most cases, the HTML is bare-bones—that is, it has only a skeleton of the web page. Usually there's a "Loading ..." message and/or rotating spinner GIF.
4. The static assets include the JavaScript code for the SPA. When loaded, this code makes additional requests for data (AJAX/XHR requests).
5. The data comes back in JSON, XML, or any other format.
6. Once the SPA receives the data, it can render missing HTML (the User Interface block in the figure). In other words, UI rendering happens on the browser by means of the SPA hydrating templates with data.[16]
7. Once the browser rendering is finished, the SPA replaces the "Loading …" message, and the user can work with the page.
8. The user sees a beautiful web page. The user may interact with the page (Inputs in the figure), triggering new requests from the SPA to the server, and the cycle of steps 2–6 continues. At this stage, browser routing may happen if the SPA implements it, meaning navigation to a new URL will trigger not a new page reload from the server, but rather an SPA rerender in the browser.

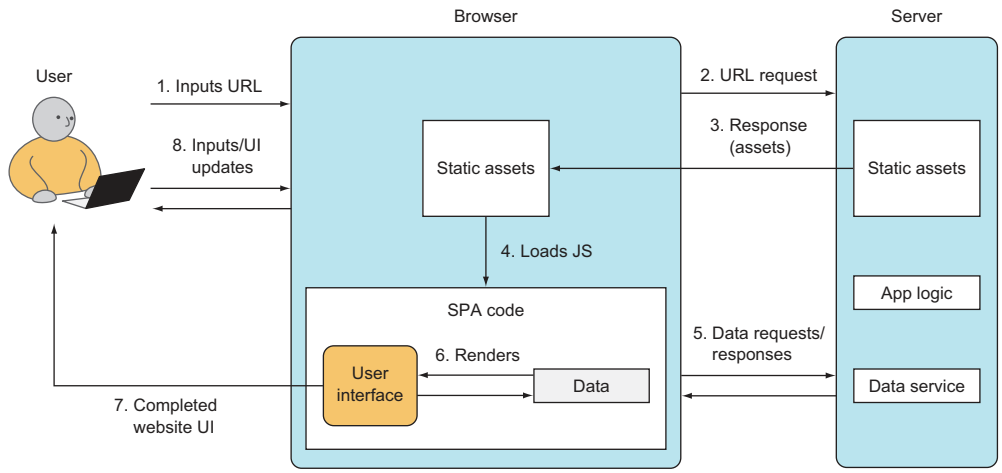

Figure 1.2 A typical SPA architecture

[16] "What does it mean to hydrate an object?" *Stack Overflow*, http://mng.bz/uP25.

To summarize, in the SPA approach, most rendering for UIs happens on the browser. Only data travels to and from the browser. Contrast that with a thick-server approach, where all the rendering happens on the server. (Here I mean *rendering* as in generating HTML from templates or UI code, not as in rendering that HTML in the browser, which is sometimes called *painting* or *drawing* the DOM.)

Note that the MVC-like architecture is the most popular approach, but it isn't the only one. React doesn't require you to use an MVC-like architecture; but, for the sake of simplicity, let's assume that your SPA is using an MVC-like architecture. You can see its possible distinct parts in figure 1.3. A navigator or routing library acts as a controller of sorts in the MVC paradigm; it dictates what data to fetch and what template to use. The navigator/controller makes a request to get data and then hydrates/populates the templates (views) with this data to render the UI in the form of the HTML. The UI sends actions back to the SPA code: clicks, mouse hovers, keystrokes, and so on.

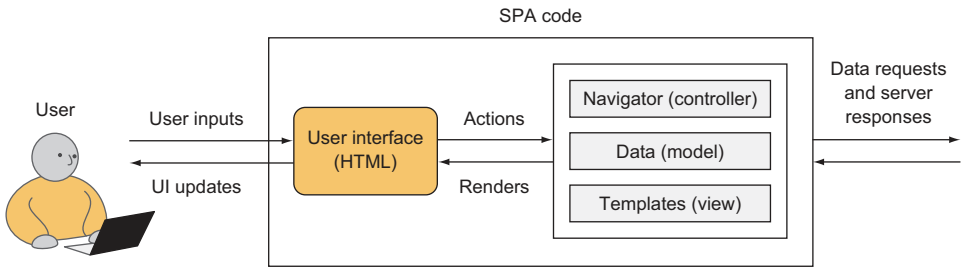

Figure 1.3 Inside a single-page application

In an SPA architecture, data is interpreted and processed in the browser (browser rendering) and is used by the SPA to render additional HTML or to change existing HTML. This makes for nice interactive web applications that rival desktop apps. Angular.js, Backbone.js, and Ember.js are examples of front-end frameworks for building SPAs.

> **NOTE** Different frameworks implement navigators, data, and templates differently, so figure 1.3 isn't applicable to all frameworks. Rather, it illustrates the most widespread separation of concerns in a typical SPA.

React's place in the SPA diagram in figure 1.3 is in the Templates block. React is a view layer, so you can use it to render HTML by providing it with data. Of course, React does much more than a typical template engine. The difference between React and other template engines like Underscore, Handlebars, and Mustache is in the way you develop UIs, update them, and manage their states. We'll talk about states in chapter 4 in more detail. For now, think of states as data that can change and that's related to the UI.

1.5.3 The React stack

React isn't a full-blown, front-end JavaScript framework. React is minimalistic. It doesn't enforce a particular way of doing things like data modeling, styling, or routing

(it's non-opinionated). Because of that, developers need to pair React with a routing and/or modeling library.

For example, a project that already uses Backbone.js and the Underscore.js template engine can switch to Underscore for React and keep existing data models and routing from Backbone. (Underscore also has utilities, not just template methods. You can use these Underscore utilities with React as a solution for a clear declarative style.) Other times, developers opt to use the *React stack*, which consists of data and routing libraries created to be used specifically with React:

- *Data-model libraries and back ends*—RefluxJS (https://github.com/reflux/refluxjs), Redux (http://redux.js.org), Meteor (https://www.meteor.com), and Flux (https://github.com/facebook/flux)
- *Routing library*—React Router (https://github.com/reactjs/react-router)
- *Collection of React components to consume the Twitter Bootstrap library*—React-Bootstrap (https://react-bootstrap.github.io)

The ecosystem of libraries for React is growing every day. Also, React's ability to describe composable components (self-contained chunks of the UI) is helpful in reusing code. There are many components packaged as npm modules. Just to illustrate the point that having small composable components is good for code reuse, here are some popular React components:

- Datepicker component: https://github.com/Hacker0x01/react-datepicker
- Set of tools to handle form rendering and validation: https://github.com/prometheusresearch/react-forms
- WAI-ARIA-compliant autocomplete (combo box) component: https://github.com/reactjs/react-autocomplete

Then there's JSX, which is probably the most frequent argument for not using React. If you're familiar with Angular, then you've already had to write a lot of JavaScript in your template code. This is because in modern web development, plain HTML is too static and is hardly any use by itself. My advice: give React the benefit of the doubt, and give JSX a fair run.

JSX is a little syntax for writing React objects in JavaScript using <> as in XML/HTML. React pairs nicely with JSX because developers can better implement and read the code. Think of JSX as a mini-language that's compiled into native JavaScript. So, JSX isn't run on the browser but is used as the source code for compilation. Here's a compact snippet written in JSX:

```
if (user.session)
  return <a href="/logout">Logout</a>
else
  return <a href="/login">Login</a>
```

Even if you load a JSX file in your browser with the runtime transformer library that compiles JSX into native JavaScript on the run, you still don't run the JSX; you run

JavaScript instead. In this sense, JSX is akin to CoffeeScript. You compile these languages into native JavaScript to get better syntax and features than that provided by regular JavaScript.

I know that to some of you, it looks bizarre to have XML interspersed with JavaScript code. It took me a while to adjust, because I was expecting an avalanche of syntax error messages. And yes, using JSX is optional. For these two reasons, I'm not covering JSX until chapter 3; but trust me, it's powerful once you get a handle on it.

By now, you have an understanding of what React is, its stack, and its place in the higher-level SPA. It's time to get your hands dirty and write your first React code.

1.6 Your first React code: Hello World

Let's explore your first React code—the quintessential example used for learning programming languages—the Hello World application. (If we don't do this, the gods of programming might punish us!) You won't be using JSX yet, just plain JavaScript. The project will print a "Hello world!!!" heading (`<h1>`) on a web page. Figure 1.4 shows how it will look when you're finished (unless you're not quite that enthusiastic and prefer a single exclamation point).

Figure 1.4 Hello World

Learning React first without JSX

Although most React developers write in JSX, browsers will only run standard JavaScript. That's why it's beneficial to be able to understand React code in pure JavaScript. Another reason we're starting with plain JS is to show that JSX is optional, albeit the de facto standard language for React. Finally, preprocessing JSX requires some tooling.

I want to get you started with React as soon as possible without spending too much time on setup in this chapter. You'll perform all the necessary setup for JSX in chapter 3.

The folder structure of the project is simple. It consists of two JavaScript files in the js folder and one HTML file, index.html:

```
/hello-world
  /js
    react.js
    react-dom.js
  index.html
```

The two files in the js folder are for the React library version 15.5.4:[17] react-dom.js (web browser DOM renderer) and react.js (React Core package). First, you need to download the aforementioned React Core and ReactDOM libraries. There are many ways to do it. I recommend using the files provided in the source code for this book, which you can find at www.manning.com/books/react-quickly and https://github.com/azat-co/react-quickly/tree/master/ch01/hello-world. This is the most reliable and easiest approach, because it doesn't require a dependency on any other service or tool. You can find more ways to download React in appendix A.

> **WARNING** Prior to version 0.14, these two libraries were bundled together. For example, for version 0.13.3, all you needed was react.js. This book uses React and React DOM version 15.5.4 (the latest as of this writing) unless noted otherwise. For most of the projects in part 1, you'll need two files: react.js and react-com.js. In chapter 8, you'll need `prop-types` (www.npmjs.com/package/prop-types), which was part of React until version 15.5.4 but is now a separate module.

After you place the React files in the js folder, create the index.html file in the hello-world project folder. This HTML file will be the entry point of the Hello World application (meaning you'll need to open it in the browser).

The code for index.html is simple and starts with the inclusion of the libraries in `<head>`. In the `<body>` element, you create a `<div>` container with the ID `content` and a `<script>` element (that's where the app's code will go later), as shown in the following listing.

Listing 1.1 Loading React libraries and code (index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
        ...
    </script>
  </body>
</html>
```

Imports the React library

Imports the ReactDOM library

Defines an empty element to mount the React UI

Starts the React code for the Hello World view

Why not render the React element directly in the `<body>` element? Because doing so can lead to conflict with other libraries and browser extensions that manipulate the

[17] v15.5.4 is the latest as of this writing. Typically, major releases like 14, 15, and 16 incorporate significant differences, whereas minor releases like 15.5.3 and 15.5.4 have fewer breaking changes and conflicts. The code for this book was tested for v15.5.4. The code may work with future versions, but I can't guarantee that it will work because no one knows what will be in the future versions—not even the core contributors.

document body. In fact, if you try attaching an element directly to the body, you'll get this warning:

```
Rendering components directly into document.body is discouraged...
```

This is another good thing about React: it has great warning and error messages!

> **NOTE** React warning and error messages aren't part of the production build, in order to reduce noise, increase security, and minimize the distribution size. The production build is the minified file from the React Core library: for example, react.min.js. The development version with the warnings and error messages is the unminified version: for example, react.js.

By including the libraries in the HTML file, you get access to the React and ReactDOM global objects: `window.React` and `window.ReactDOM`. You'll need two methods from those objects: one to create an element (React) and another to render it in the `<div>` container (ReactDOM), as shown in listing 1.2.

To create a React element, all you need to do is call `React.createElement(elementName, data, child)` with three arguments that have the following meanings:

- `elementName`—HTML as a string (for example, `'h1'`) or a custom component class as an object (for example, `HelloWorld`; see section 2.2)
- `data`—Data in the form of attributes and properties (we'll cover properties later); for example, `null` or `{name: 'Azat'}`
- `child`—Child element or inner HTML/text content; for example, `Hello world!`

Listing 1.2 Creating and rendering an `h1` element (index.html)

```
var h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  h1,
  document.getElementById('content')
)
```

Creates and saves in a variable a React element of h1 type

Renders the h1 element in the real DOM element with ID "content"

This listing gets a React element of the `h1` type and stores the reference to this object into the `h1` variable. The `h1` variable isn't an actual DOM node; rather, it's an instantiation of the React `h1` component (element). You can name it any way you want: `helloWorldHeading`, for example. In other words, React provides an abstraction over the DOM.

> **NOTE** The `h1` variable name is arbitrary. You can name this variable anything you want (such as `bananza`), as long as you use the same variable in `ReactDOM.render()`.

Once the element is created and stored in `h1`, you render it to the DOM node/element with ID `content` using the `ReactDOM.render()` method shown in listing 1.2. If you prefer, you can move the `h1` variable to the `render` call. The result is the same, except you don't use an extra variable:

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('content')
)
```

Now, open the index.html file served by a static HTTP web server in your favorite browser. I recommend using an up-to-date version of Chrome, Safari, or Firefox. You should see the "Hello world!" message on the web page, as shown in figure 1.5.

This figure shows the Elements tab in Chrome DevTools with the `<h1>` element selected. You can observe the `data-reactroot` attribute; it indicates that this element was rendered by ReactDOM.

One quick note: you can abstract the React code (listing 1.2) into a separate file instead of creating elements and rendering them with `ReactDOM.render()` all in the index.html file (listing 1.1). For example, you can create script.js and copy and paste the `h1` element and `ReactDOM.render()` call into that file. Then, in index.html, you need to include script.js after the `<div>` with ID `content`, like this:

```
<div id="content"></div>
<script src="script.js"></script>
```

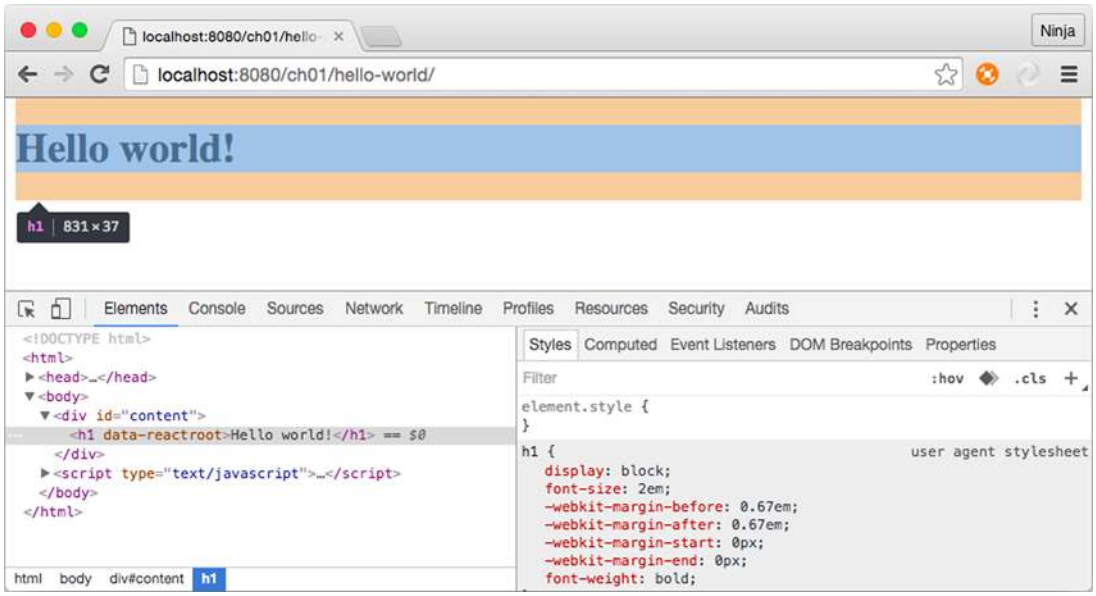

Figure 1.5 Inspecting the Hello World app as rendered by React

Local dev web server

It's better to use a local web server instead of opening an index.html file in the browser directly, because with a web server, your JavaScript apps will be able to make AJAX/XHR requests. You can tell whether it's a server or a file by looking at the URL in the address bar. If the address starts with *file*, then it's a file; and if the address starts with *http*, then it's a server. You'll need this feature for future projects. Typically, a local HTTP web server listens to incoming requests on 127.0.0.1 or localhost.

You can get any open source web server, such as Apache, MAMP, or (my favorites because they're written in Node.js) node-static (https://github.com/cloud-head/node-static) or http-server (www.npmjs.com/package/http-server). To install node-static or http-server, you must have Node.js and npm installed. If you don't have them, you can find installation instructions for Node and npm in appendix A or by going to http://nodejs.org.

Assuming you have Node.js and npm on your machine, run `npm i -g node-static` or `npm i -g http-server` in your terminal or command prompt. Then, navigate to the folder with the source code, and run `static` or `http-server`. In my case, I'm launching `static` from the react-quickly folder, so I need to put the path to Hello World in my browser URL bar: http://localhost:8080/ch01/hello-world/ (see figure 1.5).

Congratulations! You've just implemented your first React code!

1.7 Quiz

1. The declarative style of programming doesn't allow for mutation of stored values. It's "this is what I want" versus the imperative style's "this is how to do it." True or false?
2. React components are rendered into the DOM with which of the following methods? (Beware, it's a tricky question!) `ReactDOM.renderComponent`, `React.render`, `ReactDOM.append`, or `ReactDOM.render`
3. You have to use Node.js on the server to be able to use React in your SPA. True or false?
4. You must include react-com.js in order to render React elements on a web page. True or false?
5. The problem React solves is that of updating views based on data changes. True or false?

1.8 Summary

- React is declarative; it's only a view or UI layer.
- React uses components that you bring into existence with `ReactDOM.render()`.
- React component classes are created with `class` and its mandatory `render()` method.
- React components are reusable and take immutable properties that are accessible via `this.props.NAME`.

- You use pure JavaScript to develop and compose UIs in React.
- You don't need to use JSX (an XML-like syntax for React objects); JSX is optional when developing with React!
- To summarize the definition of React: React for the web consists of the React Core and ReactDOM libraries. React Core is a library geared toward building and sharing composable UI components using JavaScript and (optionally) JSX in an isomorphic/universal manner. On the other hand, to work with React in the browser, you can use the ReactDOM library, which has methods for DOM rendering as well as for server-side rendering.

1.9 Quiz answers

1. True. Declarative is a "what I want" style, and imperative is a "this is how to do it" style.
2. `ReactDOM.render`.
3. False. You can use any back-end technology.
4. True. You need the ReactDOM library.
5. True. This is the primary problem that React solves.