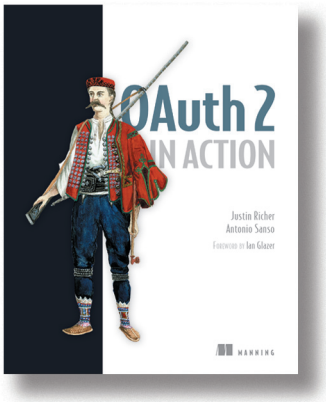
Author Picks

FREE

Exploring Modern Web Development

Chapters selected by Yakov Fain and Anton Moiseev

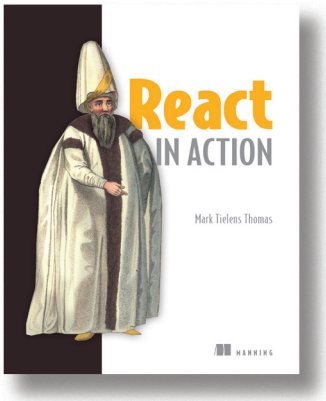
manning



OAuth 2 in Action

by Justin Richer and Antonio Sanso

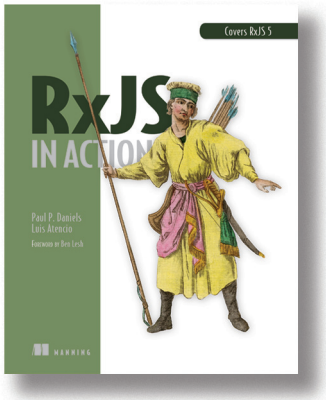
ISBN 9781617293276
360 pages
$39.99



React in Action

by Mark Tielens Thomas

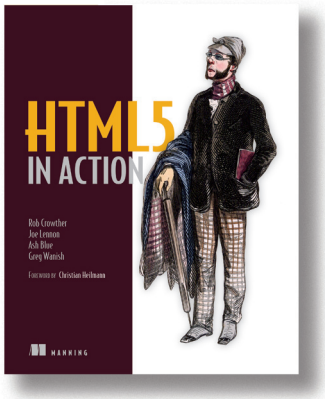
ISBN 9781617293856
360 pages
$35.99



RxJS in Action

by Paul P. Daniels and Luis Atencio

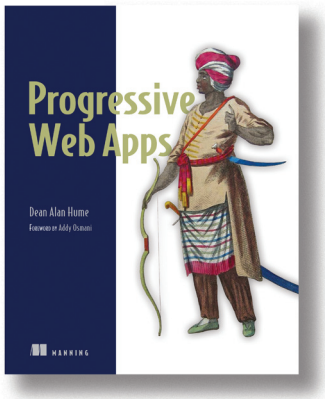
ISBN 9781617293412
352 pages
$39.99



HTML5 in Action

by Rob Crowther, Joe Lennon, Ash Blue, and Greg Wanish

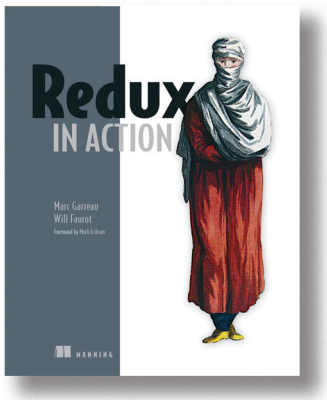
ISBN 9781617290497
466 pages
$31.99



Progressive Web Apps

by Dean Alan Hume

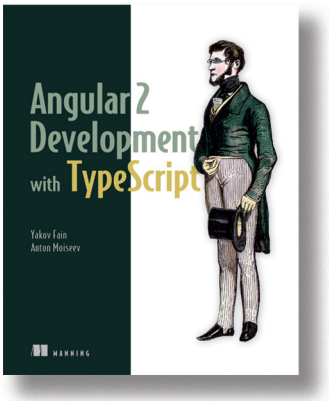
ISBN 9781617294587
200 pages
$31.99



Redux in Action

by Marc Garreau and Will Faurot

ISBN 9781617294976
312 pages
$35.99



TypeScript Quickly

by Yakov Fain and Anton Moiseev

ISBN 9781617295942
350 pages
$39.99

Exploring Modern Web Development

Chapters chosen by Yakov Fain and Anton Moiseev

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617297809

contents

introduction

To develop web applications that have all the latest and greatest features and functionality, software engineers need to be well-versed in multiple disciplines. Of course, familiarity with the JavaScript syntax is a must. As Jeff Atwood, co-founder of Stack Overflow and Discourse, says, "Any application that can be written in JavaScript *will* eventually be written in JavaScript." Well, maybe not always in the pure JavaScript, but sometimes in its more productive superset, TypeScript. This sampler, containing relevant chapters from four Manning books, gives you a taste of the multiple facets of modern web and mobile applications.

There is a trend toward developing applications using principles of reactive programming. Under this model, the data consumer subscribes to the data stream that's pushed to the consumer only when the data is available. This is an alternative to the polling model that requires the client to make periodic requests for data which may or may not be available. The RxJS library implements the push model via observable streams of data. This library offers you a variety of chainable operators (functions) that handle and transform the data en route.

In the first chapter of this guide, “Reacting with RxJS” from *RxJS in Action,* authors Paul P. Daniels and Luis Atencio explore solving problems by using a sequence of data that continuously travels from the producers to the consumers through a set of operators that implement your desired behavior—in other words, designing code with a *reactive* approach.

React is a powerful JavaScript library for building user interfaces across a variety of platforms in a declarative and component-driven way. Its mental model draws from functional and object-oriented programming, focusing on components as primary building units. The next chapter, “Meet React” from *React in Action* by Mark Tielens Thomas, introduces you to React components, some high-level concepts and paradigms, some of the tradeoffs of using React, and how teams using React can benefit as much as—or even more than— individual developers can. You’ll also take a look at one of the major pieces of technology driving React: the virtual DOM.

Next up is Chapter 1 from *OAuth 2 in Action*, "What is OAuth 2.0 and Why Should You Care?" If you've ever been asked to login to an app using a Facebook or Twitter account, then you're already acquainted with the OAuth delegation protocol. Like a valet key that gives an authorized valet limited access to your car, OAuth 2 allows the controller of a resource to grant a level of access to it from another application. In this chapter, authors Justin Richer and Antonio Sanso cover what OAuth is and what it's not. You'll also learn why authorization delegation matters and how it's used, and you'll see why OAuth is more secure and usable than the password-sharing antipattern it replaces. The OAuth security protocol is used to protect an increasing number of web APIs worldwide, and it's a worthy topic to explore when we're talking about modern web development.

The final chapter in this timely sampler is "Introducing Angular," from our own book, *Angular Development with TypeScript, Second Edition*. We'll give you an overview of Angular, a complete framework that contains everything you need to build and deploy web apps, and we'll explain why we think it's a great choice for modern web development. Together we'll build a project using the Angular CLI tool, and along the way we'll explore topics including generating bundles and JIT (just-in-time) vs. AOT (ahead-of-time) compilation.

With this varied collection of chapters, you'll appreciate the value of familiarizing yourself with multiple disciplines while getting a taste of some of the many resources available to the modern web developer. Enjoy!

In this chapter, we ask you to think in terms of streams (think reactively) and design code that, instead of holding onto data, allows data to flow through and applies transformations along the way until it reaches your desired state.

Chapter 2 from *RxJS in Action* by
Paul Pl Daniels and Luis Atencio

Reacting with RxJS

This chapter covers

- Looking at streams as the main unit of work
- Understanding functional programming's influence on RxJS
- Identifying different types of data sources and how to handle them
- Modeling data sources as RxJS observables
- Consuming observables with observers

When writing code in an object-oriented way, we're taught to decompose problems into components, interactions, and states. This breakdown occurs iteratively and on many levels, with each part further subdivided into more components, until at last we arrive at a set of cohesive classes that implement a well-defined set of interactions. Hence, in the object-oriented (OO) approach, classes are the main unit of work. Every time a component is created, it will have state associated with it, and the manipulation of that state in a structured fashion is what advances application logic. For example, consider a typical online banking website. Banking systems contain modules that encapsulate not only the business logic associated with withdrawing, depositing, and transferring money but also domain models that store and

manage other properties, such as account and user profiles. Manipulating this state (its behavior) causes the data to transform into the desired output. In other words, behavior is driven by the continuous mutation of a system's state. If such a system is designed using object-oriented programming, the units of work are the classes responsible for modeling accounts, users, money, and others.

RxJS programming works a bit differently. In reactive programming in general, the fundamental unit of work is the stream.

In this chapter, we ask you to think in terms of streams (think reactively) and design code that, instead of holding onto data, allows data to flow through and applies transformations along the way until it reaches your desired state. You'll learn how to handle different types of data sources, whether static or dynamic, as RxJS streams that use a consistent computational model based on the `Observable` data type. Unlike using other JavaScript libraries, however, using RxJS in your application means much more than implementing new APIs; it means that you must approach your problems not as the sum of the set of states manipulated by methods in classes but as a sequence of data that continuously travels from the producers to the consumers through a set of operators that implement your desired behavior.

This way of thinking places the notion of time at the forefront; this notion runs as the undercurrent through the components of an RxJS stream and causes data to be never stored but rather transiently flowing. Relating this to a real-world physical water stream, you can think of the data source as the top of the stream and the data consumer as the bottom of the stream. Hence, data is *always traveling downstream*, in a single direction, like water in a river, and along the way you can have control dams in charge of transforming the nature of this stream. Thinking this way will help you understand how data should move through an application.

This is not to say that this understanding will come easily—like any new skill, it must be built up over time and through iterative application of the concepts. As you saw in the pseudo streams example in chapter 1, the notion of data in motion versus data kept in variables is a difficult one for most people to wrap their head around. In this book, we'll provide you with the necessary tools to ease this learning curve. To begin building your toolkit, this chapter lays the groundwork to help you better understand streams. Many of the basic principles behind RP derive from functional programming, so let's start there.

2.1 Functional programming as the pillar of reactive programming

The abstractions that support RP are built on top of FP, so FP is the foundation for RP. Much of the hype around RP derives from the development communities and the industry realizing that FP offers a compelling way to design your code. This is why it's important for you to have at least a basic understanding of the FP principles. If you have a solid background in functional programming, you're free to skip this section, but we recommend you read along because it will help you better understand some of the design decisions behind RxJS.

Just like in chapter 1, we ask you to take another quick glance at the main website for the Reactive Extensions project (http://reactivex.io). In it, you'll find the following definition:

> *ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.*

You learned about the main components of the observer pattern in chapter 1 (producer and consumer); now you'll learn the about the other parts that gave rise to the Rx project, which are functional programming and iterators. Here's a diagram (figure 2.1) that better illustrates the relationship between these paradigms.

Let's begin by exploring the basics of FP.

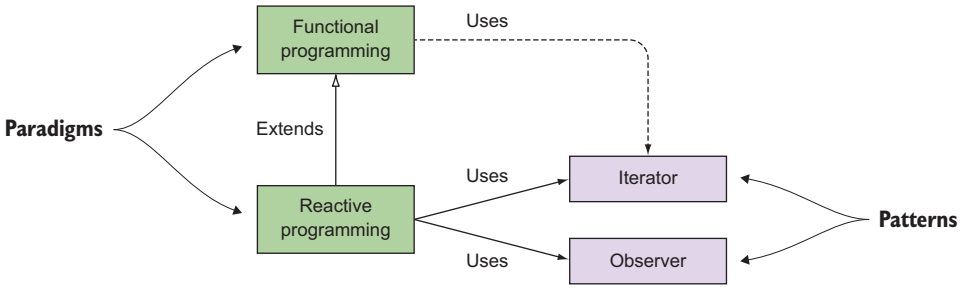


Figure 2.1 The RP paradigm builds and extends from FP. Also, it leverages commonly known design patterns such as iterator and observer.

2.1.1 Functional programming

Functional programming is a software paradigm that emphasizes the use of functions to create programs that are declarative, immutable, and side effect–free. Did you trip over the word *immutable*? We agree with you; the notion of a program that doesn't ever change state is a bit mind bending. After all, that's why we put data in variables and modify them to our heart's content. All of the object-oriented or procedural application code you've written so far relies on changing and passing variables back and forth to solve problems. So how can you accomplish the same goals without doing this? Take the example of a clock. When a clock goes from 1:00 p.m. to 2:00 p.m., it's undoubtedly changing, isn't it? But to frame this from a functional point of view, we argue that instead of a single clock instance mutating every second, it's best to return new clock instances every second. Theoretically, both would arrive at the same time, and both would give you a single state at the end.

RxJS borrows numerous principles from FP, particularly *function chaining*, *lazy evaluation*, and the notion of using an abstract data type to orchestrate data flows. These are some of the design decisions that drive the development of RxJS's stream programming via the `Observable` data type. Before we dive in, we'll explain the main

parts of the FP definition we just gave and then show you a quick example involving arrays.

To reiterate, functional programs have the following characteristics:

- *Declarative*—Functional code has a peculiar style, which takes advantage of JavaScript's higher-order functions to apply specialized business logic. As you'll see later on, function chains (also known as *pipelines*) describe data transformation steps in an idiomatic manner. Most people see SQL syntax as a perfect example of declarative code.
- *Immutable*—An immutable program (and by this we mean any immutable function, module, or whole program) is one that never changes or modifies data after it's been created or after its variables have been declared. This can be a radical concept to grasp, especially when you're coming from an OO background. Functional programs treat data as immutable, constant values. A good example of a familiar module is the `String` type, because none of the operations change the string on which they operate; rather, they all return new strings. A good practice that you'll see us use throughout the book is to qualify all of our variables with `const` to create nicely block-scoped immutable variables that can't be reassigned. This doesn't solve all the problems of immutability, but it gives you a little extra support when your data and functions are shared globally.
- *Side effect–free*—Functions with side effects depend on data residing outside its own local scope. A function's scope is made up of its arguments and any local variables declared within. Interacting with anything outside this (like reading a file, writing to the console, rendering elements on an HTML page, and more) is considered a side effect and should be avoided or, at the very least, isolated. In this book, you'll learn how RxJS deals with these issues by pushing the effectful computations into the subscribers.

In general, mutations and side effects make functions unreliable and unpredictable. That is to say, if a function alters the contents of an object inadvertently, it will compromise other functions that expect this object to keep its original state. The OO solution to this is to encapsulate state and protect it from direct access from other components of the system. In contrast, FP deals with state by eliminating it, so that your functions can confidently rely on it to run.

For instance, figure 2.2 illustrates the dependency between the two functions `doWork()` and `doMoreWork()` through a shared state variable called `data`.

This coupling presents an issue because `doMoreWork` now relies on `doWork` to run first. Two issues may occur:

- The result of `doMoreWork()` depends entirely on the successful outcome of `doWork()` and on no other parts of the system changing this variable.
- Unit tests against this function can't be done in isolation as they should be, so your test results are susceptible to the order in which the test cases are run (in chapter 9, we'll explore testing in much more detail).

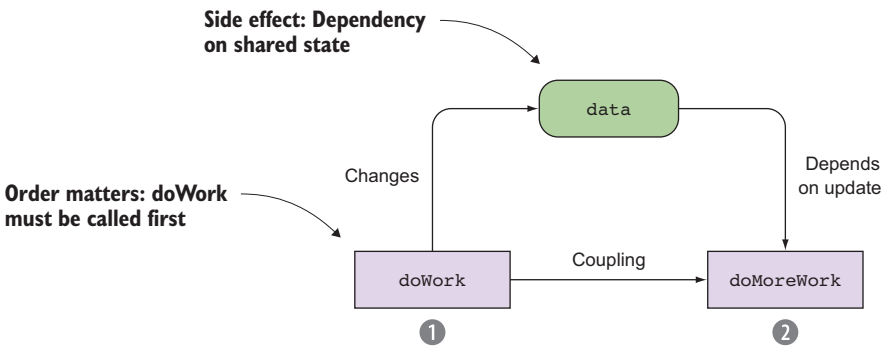


Figure 2.2 Function `doWork()` is temporarily coupled to `doMoreWork()` because of the dependency on shared state (side effect). Hence, `doWork()` must be called before `doMoreWork()` or the program will cease to work.

Shared variables, especially in the global scope, add to the cognitive load of reasoning about your code because these variables demand that you keep track of them as you trace through it. Another way you can think of global data is as a hidden parameter within all your functions. So the more global the state you have to maintain, the harder it is for you to maintain your code. The example in figure 2.2 is an obvious side effect, but they're not always this clear. Consider this trivial function that returns the lowest value in a numerical array:

```
const lowest = arr => arr.sort().shift();
```

Although this code may seem harmless to you, it packs a terrible side effect. Can you spot it? This function actually changes the contents of the input array, as shown in the following snippet. So if you used the first element of the array somewhere else, that's completely gone now:

```
let source = [3,1,9,8,3,7,4,6,5];
let result = lowest(source); //-> 1
console.log(source); //-> [3, 3, 4, 5, 6, 7, 8, 9]
```

The original array changed!

Later on, we'll talk about a functional library that provides a rich set of functions for working with arrays immutably, so that things like this don't inadvertently creep up on you.

Matters get worse if you have concurrent asynchronous processes where data structures are shared and used in different components. Because latency is unpredictable, you'd need to either nest your function calls or use some other robust synchronization mechanism to ensure they execute and mutate this state in the right order; otherwise, you'll experience random and hard-to-troubleshoot bugs.

Fortunately, JavaScript is single threaded, so you don't need to worry about shared state running through different threads. But as JavaScript developers, we deal quite often with concurrent code when either working with web workers or making simultaneous

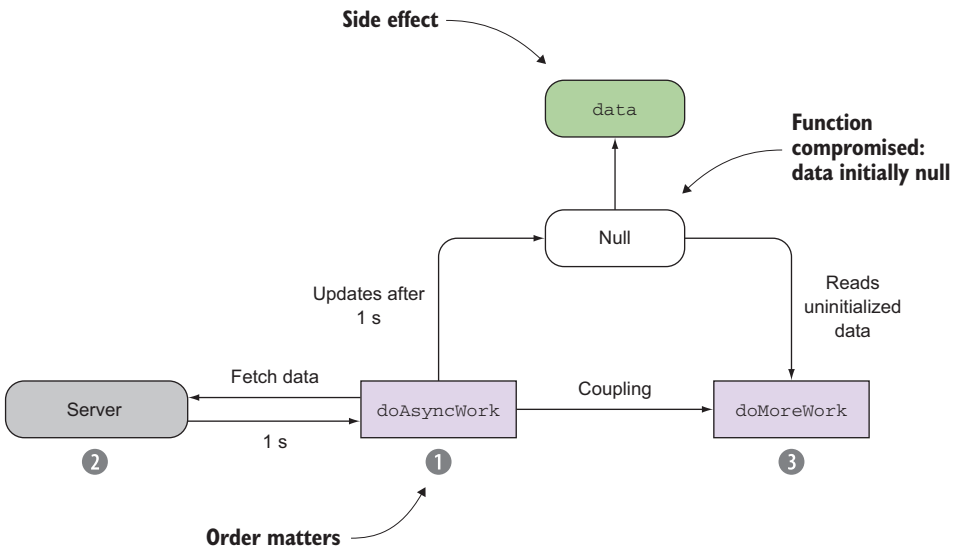


Figure 2.3 **Function `doAsyncWork()` is an example of a remote call that fetches data from the server. Suppose this call has a latency around one second, depending on network conditions. Immediately after, the next function runs `doMoreWork()`, expecting that a piece of shared data has already been initialized. Because of this latency, the shared data has not been initialized, and the execution of `doMoreWork()` is compromised.**

HTTP calls. Consider the trivial yet frequent use case illustrated in figure 2.3, which involves asynchronous code mixed with synchronous code. This presents a tremendous challenge because the latter assumes that the functions executing before it have completed successfully, which might not necessarily be the case if there's some latency.

In this scenario, `doAsyncWork()` fetches some data from the server, which never completes in a constant amount of time. So `doMoreWork()` fails to run properly because it reads data that hasn't yet been initialized. Callbacks and `Promises` help you solve this problem, so that you don't have to hardcode your own timeouts in order to anticipate latency. Dealing directly with time is a recipe for disaster because your code will be extremely brittle and hard to maintain and will cause to you to come in to work during a weekend when your application is experiencing slightly more traffic than usual. Working with data immutably, using FP, and the help of an asynchronous library like RxJS can make these timing issues disappear—immutable variables are protected against time. In chapters 4 and 6, we'll cover timing and synchronization with observables, which offer a much superior solution to this problem.

Even though JavaScript isn't a pure functional language, with a bit of discipline and the help of the proper libraries you can use it completely functionally. As you learn to use RxJS, we ask that you also begin to embrace a functional coding style; it's something we believe strongly about and promote in all code samples in this book.

Aside from using `const` to safeguard the variable's reference, JavaScript also has support for a versatile array data structure with methods such as `map`, `reduce`, `filter`, and others. These are known as higher-order or first-class functions, and they're one of the most important functional qualities in the language, allowing you to express JavaScript programs in an idiomatic way. A higher-order function is defined as one that can accept as argument as well as return other functions; they're used extensively with RxJS, as with any functional data type.

The following listing shows a simple program that takes an array of numbers, extracts the even numbers, computes their squares, and sums their total.

Listing 2.1 Processing collections with `map`, `reduce`, and `filter`

```
const isEven = num => num % 2 === 0;
const square = num => num * num;
const add = (a, b) => a + b;

const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

arr.filter(isEven).map(square).reduce(add);  //-> 220
```

In this example, because these operations are side effect–free, this program will always produce the same value (`220`), given the same input array.

Where can I find this code?

All the code for this book can be found in the RxJS in Action GitHub repository at https://github.com/RxJSInAction. There, you'll find two subrepositories. Under rxjs-in-action, you'll find a simple application that contains the code for all individual chapter listings for chapters 1 through 9. All samples are presented as runnable snippets of RxJS code that you can interact with. Also, under the banking-in-action repository, you'll find our web application that showcases RxJS embedded into a React/Redux architecture. Some of the APIs that we interact with in the book don't allow cross-origin resource sharing (CORS). The simplest way to get around this is to disable it at the browser level by installing an extension or add-on.

If you imagine for a second having to write this program using a non-functional or imperative approach, you'll probably need to write a loop, a conditional statement, and a few variables to keep track of things. FP, on the other hand, raises the level of abstraction and encourages a style of declarative coding that clearly states the purpose of a program, describing *what* it does and not *how* it does it. Nowhere in this short program is the presence of a loop, `if/else`, or any imperative control flow mechanism.

One of the main themes in FP that you'll use as well in RP is *programming without loops*. In listing 2.1, you took advantage of `map`, `reduce`, and `filter` to hide manual looping constructs—allowing you to implement looping logic through functions' arguments. Moreover, these functions are also immutable, which means that new arrays are created at each step of the way, keeping the original intact.

Going back to our discussion, side effect–free functions are also known as *pure*, because they're predictable when you're working on collections of objects or streams. You should always strive for purity whenever possible because it makes your programs easy to test and reason about.

Want to learn more about functional programming?

JavaScript's `Array` object has a special place in functional programming because it behaves as an extremely powerful data type called a *functor*. In a simple sense, functors are containers that can wrap data and expose a mapping method that allows you to immutably apply transformations on this data, as shown by the `Array.map()` method. As you'll see learn later on, RxJS streams follow this same functor-like design.

Functional programming is a huge subject to cover. In this book, we'll cover only enough of FP to help you to understand and be proficient with RxJS and RP. If you'd like more information about FP and FP topics, you can read about them in detail in *Functional Programming in JavaScript* (Manning, 2016) by Luis Atencio.

The code shown in listing 2.1, which works well with arrays, also translates to streams. Along the lines of the pseudo `Stream` data type that we discussed in chapter 1, look at how similarly arrays and streams work when processing some number sequence:

```
Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .filter(isEven)
  .map(square)
  .reduce(add)
  .subscribe(console.log);  //-> 220
```

You can clearly see how Rx was inspired by FP. All we had to do was wrap the array into a stream and then subscribe to it to listen for the computed values that derive from the sequence of steps declared in the stream's pipeline. This is the same as saying that streams are containers that you can use to lift data (events) into their context, so that you can apply sequences of operations on this data until reaching your desired outcome. Fortunately, you're already familiar with this concept from working with arrays for many years. You can lift a value into an array and map any functions to it. Suppose you declare some simple functions on strings like `toUpper`, `slice`, and `repeat`:

```
['rxjs'].map(toUpper).map(slice(0, 2)).map(repeat(2)); //-> 'RXRX'
```

The ancient Greek philosopher Heraclitus once said, "You can never step into the same river twice." He formulated this statement as part of his doctrine on *change* and *motion* being central components of the universe—everything is constantly in motion. This epic realization is what RxJS streams are all about: as data continuously flows and moves through the stream, orchestrated through this is the data type you're learning about called `Stream`. Despite being dynamic, `Streams` are immutable data types. Once

a `Stream` is declared to wrap an array, listen for mouse clicks, or respond to an HTTP call, you can't mutate it or add a new value to it afterward—you must do it at the time of declaration. Hence, you're *specifying the dynamic behavior of an object or value declaratively and immutably*. We'll revisit this topic a bit more in the next chapter.

Moreover, the business logic of this program is pure and takes advantage of side effect–free functions that are mapped onto the stream to transform the produced data into the desired outcome. The advantage of this is that all side effects are isolated and pushed onto the consumers (logging to the console, in this case). This separation of concerns is ideal and keeps your business logic clean and pure. Figure 2.4 shows the role that the producers and consumers play.

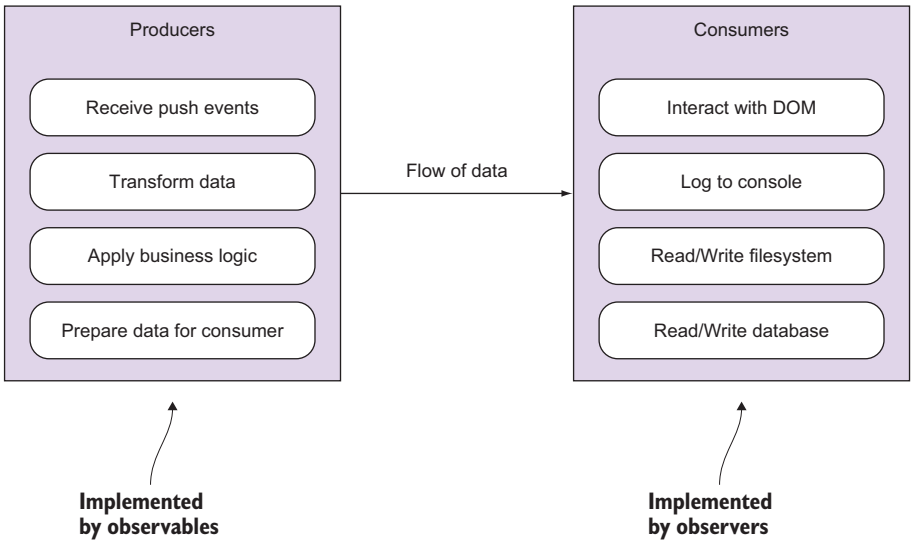


Figure 2.4 Events emitted by producers are pushed through a pipeline of side effect–free functions, which implement the business logic of your program. This data flows to all observers in charge of consuming and displaying it.

Another design principle of streams that's borrowed from FP is lazy evaluation. *Lazy evaluation* means that code is never called until actually needed. In other words, functions won't evaluate until their results are used as part of some other expression. In the following example, the idea is that a stream sits idle until a subscriber (a consumer) is attached to it; only then will it emit the values `1–10`:

```
Stream([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
  .filter(isEven)
  .map(square)
  .reduce(add);
```

Nothing runs here because no subscriber is added.

When a subscriber begins listening, the stream will emit events downstream through the pipeline in a single, unidirectional flow from the producer to the consumer. This

is beneficial if your functions have side effects because the pipeline runs in a single direction, helping to ensure an orderly execution of your function calls. This is another reason to avoid side effects at all costs, especially when you begin combining multiple streams, because things can revert into the tangled mess that you're trying to get rid of in the first place. Lazy evaluation is a mandatory requirement for streams because they emit data infinitely to handle mouse movements, key presses, and other asynchronous messages. Otherwise, storing the entire sequence of mouse movements in memory could make your programs crash.

Reactive Manifesto

One of the key principles of a reactive system is the ability to stay afloat under varying workloads—known as *elasticity*. Obviously, this has many architectural and infrastructural implications that extend beyond the scope of this book, but a corollary to this is that the paradigm you use shouldn't change whether you're dealing with one, one hundred, or thousands of events. RxJS offers a single computing model to handle finite as well as infinite streams.

The Reactive Manifesto (http://www.reactivemanifesto.org) was published by a working group that aims at identifying patterns for building reactive systems. It has no direct relation to the Rx libraries, but philosophically there are many points in common.

For instance, without lazy evaluation, code that uses infinite streams like this will cause the application to run out of memory and halt:

```
//1
Stream.range(1, Number.POSITIVE_INFINITY)
  .take(100)
  .subscribe(console.log);
```

Reads infinitely many numbers in memory

```
//2
Stream.fromEvent('mousemove')
   .map(e => [e.clientX, e.clientY])
   .subscribe(console.log);
```

Listens to all mouse moves the user is performing

In example 1, lazy evaluation makes the stream smart enough to understand that it will never need to actually run through all the positive numbers infinitely before taking the first 100. And even if the amount of numbers to store is big, streams won't persistently hold onto data; instead, any data emitted is immediately broadcast to all subscribers at the moment it gets generated. In example 2, imagine if you needed to store in memory the coordinates of all mouse movements on the screen; this could potentially take up a huge amount of memory. Instead of holding onto this data, RxJS lets it flow freely and uses the iterator pattern to traverse any type of data source irrespective of how it's created.

2.1.2 The iterator pattern

A key design principle behind RxJS streams is to give you a familiar traversal mechanism, just as you have with arrays. Iterators are used to traverse containers of data in a structure-agnostic way or independent of the underlying data structure used to harness these elements, whether it's an array, a tree, a map, or even a stream. In addition, this pattern is effective at *decoupling the business logic applied at each element from the iteration itself*. The goal is to provide a single protocol for accessing each element and moving on to the next, as shown in figure 2.5.

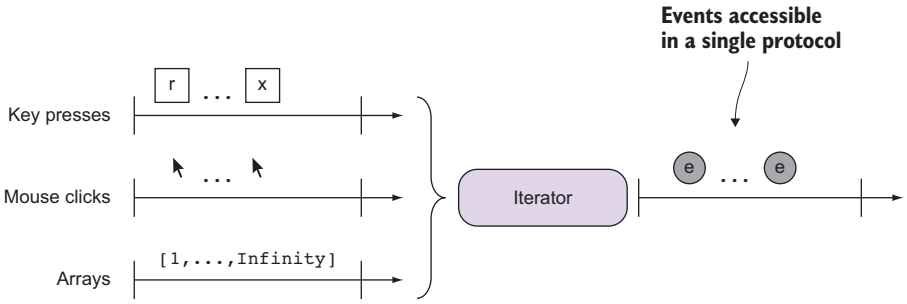


Figure 2.5 Iterators abstract the traversal mechanism, whether a `for` or a `while` loop, so that processing any type of data is done in the exact same way.

We'll explain this pattern briefly now, and later on you'll see how this applies to streams. The JavaScript ES6 (or ES2015) standard defines the iterator protocol, which allows you to define or customize the iteration behavior of any iterable object. The iterable objects you're most familiar with are arrays and strings. ES6 added `Map` and `Set`. With RxJS, we'll treat streams as iterable data types as well.

You can make any object iterable by manipulating its underlying iterator. We'll be using some ES6-specific syntax to show this. Consider an iterator object that traverses an array of numbers and buffers a set amount of contiguous elements. Here, the business logic performed is the buffering itself, which can be useful to group elements together to form numerical sets of any dimension, like the ones illustrated in figure 2.6.

Now let's see what the code would look like. The next listing shows the internal implementation of this custom iterator, which contains the buffer logic.

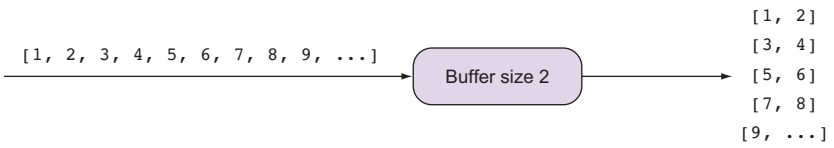


Figure 2.6 Using an iterator to display sets of numbers of size 2

Listing 2.2 Custom `BufferIterator` function

```
function BufferIterator(arr, bufferSize = 2) {
    this[Symbol.iterator] = function () {
        let nextIndex = 0;

        return {
            next: () => {
              if(nextIndex >= arr.length) {
                return {done: true};
              }
              else {
                let buffer = new Array(bufferSize);
                for(let i = 0; i < bufferSize; i++) {
                  buffer[i] = (arr[nextIndex++]);
                }
                return {value: buffer, done: false};
              }
            }
        }
    };
}
```

Assigns a default buffer size of 2

Overrides the provided array's iterator mechanism. Symbol.iterator represents the array's iterator function.

The next() function is part of the Iterator interface and marks the next element in the iteration.

Returns an object with a done = true property, which causes the iteration mechanism to stop

Creates a temporary buffer array to group contiguous elements

Returns the buffered items and a status of done = false, which indicates to the iteration mechanism to continue

Any clients of this API need only interact with the `next()` function, as outlined in the class diagram in figure 2.7. The business logic is hidden from the caller, the `for...of` block, which is the main goal of the iterator pattern.

The `next()` function in listing 2.2 is used to customize the behavior of the iteration through `for...of` or any other looping mechanism. As you'll see later on, RxJS observers also implement a similar interface to signal to the stream to continue emitting elements.

DID ITERATORS THROW YOU FOR A LOOP? The ES6 iterator/iterable protocols are powerful features of the language. RxJS development predates this protocol, so it doesn't use it at its core, but in many ways the pattern is still applied. We don't use iterators in this book; nevertheless, we recommend you learn about them. You can read more about this protocol here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#iterator.

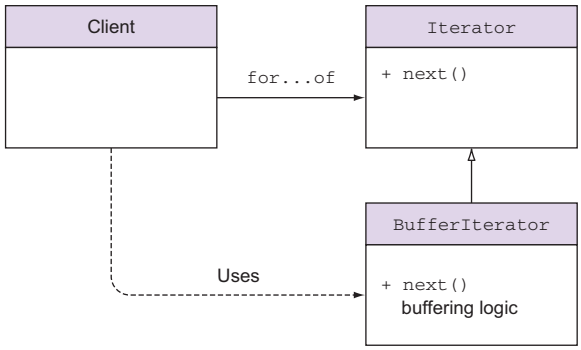


Figure 2.7 A class diagram (UML) highlighting the components of the iterator pattern. The `Iterator` interface defines the `next()` function, which is implemented by any concrete iterator (`BufferIterator`). Users of this API need only interact with the interface, which is general and applies to any custom traversal mechanism.

Iterators allow you to easily take advantage of the JavaScript runtime to take care of the iteration on your behalf. Following, we show some examples of this using our simple numerical domain. Buffering is built into RxJS, and it's really useful to gather up a sequence of events and make decisions about the nature of these events or apply additional logic. An example of this is when you need to invoke an expensive operation in response to a sequence of mouse events, like drag and drop. Instead of running expensive code at each mouse position, you buffer a specific number of them and emit a single response, taking all into account. Implementing this yourself would be tricky, because it would involve time management and keeping external state that tracks the frequency and speed with which the user moves the mouse; certainly, you'll want to delegate this to libraries that understand how to manage all this for you. We'll examine buffers in more detail in chapter 4. In RxJS, buffers aren't implemented as in listing 2.2, but it serves to show you an example of how you can buffer data using iterators, which is how you think about these sorts of operations. Here's our `BufferIterator` in action:

```
const arr = [1, 2, 3, 4, 5, 6];

for(let i of new BufferIterator(arr, 2)) {
    console.log(i);
}
//-> [1, 2] [3, 4] [5, 6]

for(let i of new BufferIterator(arr, 3)) {
    console.log(i);
}
//-> [1, 2, 3] [4, 5, 6]
```

Buffers two elements at once

Buffers three elements at once. Notice how the iteration mechanism is completely separate from the buffering logic.

When you subscribe to a stream, you'll be traversing through many other data sources such as mouse clicks and key presses in the exact same way. Theoretically speaking, because our pseudo `Stream` type is an iterable object, you could traverse a set of key press events as well with a conventional loop:

```
const stream = Stream(R, x, J, S)[Symbol.iterator]();

for(let keyEvent of stream) {
    console.log(event.keyCode);
}
//-> 82, 120, 74, 83
```

Creating a stream that wraps key presses for those four letters

Traversing a stream is semantically equivalent to subscribing to it (more on this later).

Streams in RxJS also respect the `Iterator` interface, and subscribers of this stream will listen for all the events contained inside it. As you saw previously, iterators are great at decoupling the iteration mechanism and data being iterated over from the business logic. When data defines the control flow of the program, this is known as *data-driven code*.

2.2 Stream's data-driven approach

RxJS encourages a style of development known as data-driven programming. The data-driven approach is a way of writing code such that you can separate the behavior of an application from the data that's passing through it. This is a core design decision of RxJS and the main reason why you can use the same paradigm to process arrays, mouse clicks, or data from AJAX calls.

In the OO approach, you place more emphasis on the supporting structures than the data itself. This explains why pure OO languages like Java have many different implementations to store a sequential collection of elements, each tackling different use cases: `Array`, `ArrayList`, `LinkedList`, `DoublyLinkedList`, `ConcurrentLinkedList`, and others. To put it another way, imagine that you run a local florist that performs deliveries. Your business in this case is importing flowers, cutting them, packaging them, handling orders, and sending those orders out for delivery. These tasks are all part of your business logic; that is, they are the important bits that your customers care about and the parts that bring in revenue. Now imagine that in addition to those tasks, you're also tasked with designing the type of delivery van to use. Creating this structure is itself a full-time job and one that would likely distract from your primary business without meaningfully lending to it.

Data, as in the data that you care about and that which gives rise to search engines, websites, and video games, is the flower component of software design. Creating software should therefore be about how you manipulate data rather than how you create approximations of real-world objects (as you might in OO programming). Bringing data to the forefront and separating it from the behavior of the system is at the heart of data-driven/data-centric design. Similarly, loosely coupling functions from the objects that contain data is a design principle of FP and, by extension, RP.

To be driven by data is to be compelled to act by the presence of it and to let it fuel your logic. Without data to act on, behavior should do nothing. The idea of data giving life to behavior ties back to our earlier definition of what it means to be reactive—reacting to data instead of waiting for it. Streams are nothing more than a passive process that sits idle when nothing is pushed through them and no consumer is attached, as shown in figure 2.8.

This design pattern seems intuitive to most people because we think of data as requiring some sort of behavior in order to be meaningful. In a physics simulation, the mass of a ball is just a decimal number without context until the behavior of gravity is applied to it. Thus, if we are to imagine that both are intertwined by nature, it seems only natural that they should cohabitate logically within an object. In theory, this would seem to be a fairly obvious approach, and indeed the prevalence and popularity of OO programming stands testament to its power as a programming paradigm.

But it turns out that the greatest strength of OO design is also perhaps its greatest weakness. The intuition of representing components as objects with intrinsic behavior makes sense to a certain extent, but much like the real world, it can become difficult to reason about as the complexity of the application grows. For instance, if you hadn't

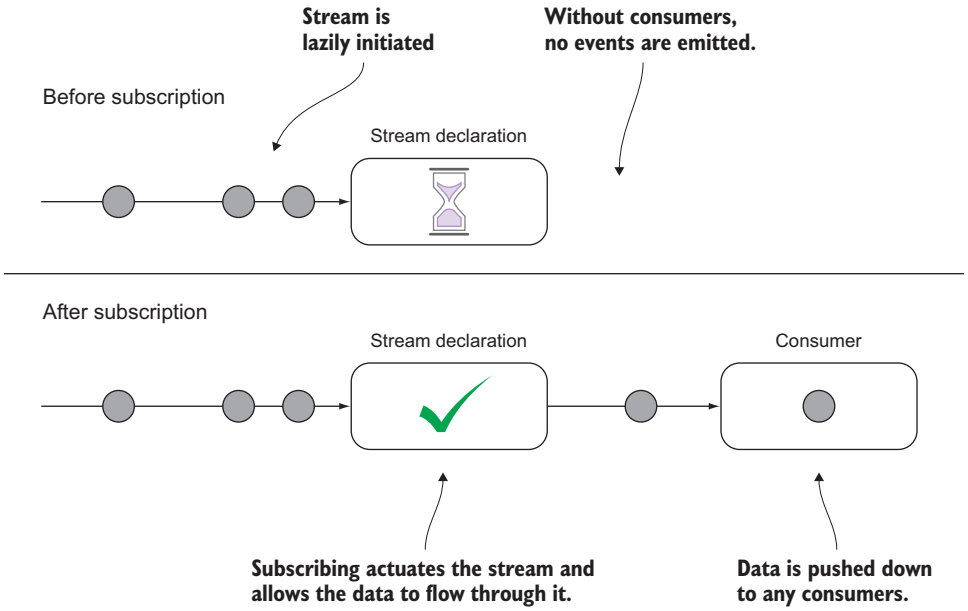


Figure 2.8 **Initially, streams are lazy programs that wait for a subscriber to become available. Any events received at this point are discarded. Subscribing to the stream puts the wheels in motion, and event data flows through the pipeline and out for consumers to use.**

used the `BufferIterator` type before, you would've had to implement the buffering logic with the application logic that uses this data. To keep things simple, you just logged the numbers to the screen, but in real life you'll use iterators for something more meaningful.

The data-centric approach seeks to remedy this issue by separating the concerns of data and behavior, through its producer/consumer model. Data would be lifted out of the behavior logic and instead would pass through it. Behavior could be loosely linked such that the data moved from one part of the application to another, independent of the underlying implementation. Earlier you saw how iterators help with this:

```
Stream([1, 2, 3, 4, 5, 6])
   .buffer(2)
   .subscribe(console.log)); //-> [1, 2] [3, 4] [5, 6]
```

Each step in the pipeline resides within its own scope that's externalized from the rest of the logic. In this case, you can see that just like iterators, the buffering step is done separately from the code acting on the data. By constructing it so, you've both declared the intent of each step and effectively decoupled the data from the underlying implementation, because each component reacts only to the step that preceded it.

Furthermore, producers come in all shapes and sizes. Event emitters are one of the most common ones; they're used to respond to events like mouse clicks or web requests. Also, there are timer-based sources like `setTimeout` and `setInterval` that

will execute a task at a specified point in the future. There are subtler ones such as arrays and strings, which you might recognize as collections of data but not necessarily producers of data.

Traditionally, when dealing with each of these data sources, you've been conditioned to think of them as requiring a different approach. For instance, event emitters require named event handlers, `Promises` require the continuation-passing "thenable" function, `setTimeout` needs a callback, and arrays need a loop in order to iterate through them. What if we told you that all of these data sources can be consolidated and processed in the exact same way?

2.3 Wrapping data sources with Rx.Observable

All along, we've been using a pseudo data type called `Stream` as a substitute for the real `Rx.Observable` type available in RxJS 5. We did this to help you understand the paradigm and what it means to think in streams, rather than focus on the specifics of the library. In this section, we'll begin diving into the RxJS 5 APIs (for information about installing RxJS 5 on the client or on the server, please visit appendix A). Through the `Rx.Observable` type, you can subscribe to events produced from different types of data sources.

> **ES7 SPECIFICATION** One of the key design decisions behind the development of RxJS 5 was to create an `Observable` type that follows the proposed observable specification slated for the next version of JavaScript ES7. You can find all the details of this API here: https://github.com/zenparsing/es-observable.

You can lift a heterogeneous set of inputs into the context of an observable object. Doing so allows you to unlock the power of RxJS to transform or manipulate them to reach your desired outcome. First, let's identify these different types of data.

2.3.1 Identifying different sources of data

We mentioned earlier that the advantage of separating data and behavior is that you can reason about a holistic model to account for any type of data. Hence, the first step to break the data free is to understand that all of these data sources are the same when viewed through a data-driven (or stream-driven?) lens. First, let's re-categorize the types of data we'll encounter. Rather than dealing with them as strict JavaScript types, let's look at some broader categories of data.

EMITTED DATA

Emitted data is data that will be created as a result of some sort of interaction with the system; this can be either from a user interaction such as a mouse click or a system event like a file read. As we alluded to in chapter 1, some of these will have at most one event; that is, you request data and then, at some point in the future, you receive a response. For this, `Promises` can be a good solution. Others, like a user's clicks and key presses, are part of a continuous process, and this requires you to treat them as event emitters that produce multiple discrete events at future times.

STATIC DATA

Static data is data that's already in existence and present in the system (in memory); for example, an array or a string. Artificial unit test data also falls into this category. Interacting with it is usually a matter of iterating through it. If you were wrapping a stream around an array, for instance, the stream would never actually store the array; it would extend it with a mechanism that flushes the elements within the array (based on iterators). Arrays are a common and heavily used static data source, but you could also think of associative arrays or maps as unordered static data. Most of the examples so far have dealt with static data such as strings, numbers, and arrays, which we used to illustrate some of the basic concepts. In later parts of the book, we'll focus on emitted data and generated data.

GENERATED DATA

Generated data is data that you create periodically or eventually, like a clock sounding a chime every quarter hour; it can also be something more procedural like generating the Fibonacci sequence using ES6 generators. In the latter case, because the sequence is infinite, it's not feasible to store it all in memory. Instead, each value should be generated on the fly and yielded to the client as needed. In this category, you can also place the traditional `setTimeout` and `setInterval` functions, which use a timer to trigger events in the future.

Just like the saying, "When you're a hammer, every problem looks like a nail," the `Rx.Observable` data type can be used to normalize and process each of these data sources using a single programming model—it's the hammer. With this approach, you gain the most code reuse and avoid creating specific ad hoc functions to deal with the idiosyncrasies of each event type.

2.3.2 Creating RxJS observables

In Rx, an observer subscribes to an observable. As you learned in chapter 1, this is analogous to the observer pattern with the subject acting as the observable; `Rx.Observable` represents the object that pushes notifications for observers to receive. The observers asynchronously react to any events emitted from the observable, which allows your application to remain responsive instead of blocking in the face of a deluge of events. This is ideal to implement asynchronous, responsive code both on the client and on the server.

`Rx.Observable` has different meanings to different people. To functional programming purists, it falls under a special category called a functor, an *endofunctor* to be exact. (We don't cover functors in this book because they're not essential to understanding Rx, but if you want learn more about them, you'll find them in the functional programming book mentioned earlier.) To most others, it's simply a data type that wraps a given data source, present in memory or eventually in the future, and allows you to chain operations onto it by invoking observable instance methods sequentially. Figure 2.9 shows a simple visualization of this concept.

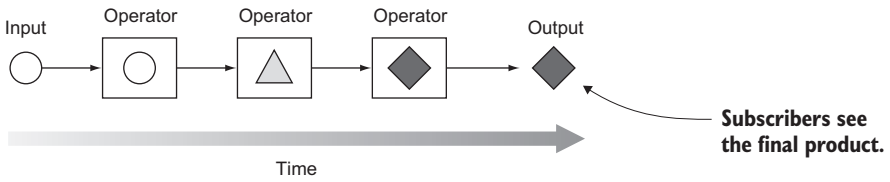


Figure 2.9 The sequential application of methods or operators that transform an input into the desired outcome, which is what subscribers see

Here's a quick look at how observables implement chaining extremely well:

```
Rx.Observable.from(<data-source>)
    .operator1(...)
    .operator2(...)
    .operator3(...)
    .subscribe(<process-output>);
```

Wraps a data source with a stream

Invokes a sequence of operations chained by the dot operator. In chapter 3, we'll spend a lot more time with observable instance methods.

Processes the results

Whether you choose to accept one definition over the other, it's important to understand that an observable doesn't just represent a value *now* but also the *idea of a value occurring in the future*. In FP, this is the same definition given to pure functions, which are nothing more than to-be-computed values, and part of the reason why we refer to the "methods" invoked on an observable instance as *operators*.

Because observables in RxJS are immutable data types, this pattern works quite well and should not look that foreign to you. Consider a familiar data type, `String`. Look at this trivial example and notice its similarity to the previous pattern:

```
String('RxJS')
    .toUpperCase()
    .substring(0, 2)
    .concat(' ')
    .repeat(3)
    .trim()
    .concat('!') //->  "RX RX RX!"
```

Learning about a shiny new tool is always exciting, and there's a tendency among developers to try to use that tool in every conceivable situation where it might potentially apply. But as is often the case, no tool is meant for every situation, and it's just as important to understand where RxJS won't be used.

You can divide your computing tasks into four groups within two different dimensions. The first dimension is the number of pieces of data to process. The second is the manner in which the data must be processed, that is, synchronously or asynchronously. In enumerating these possibilities, we want to highlight where RxJS would be most beneficial to your applications.

2.3.3 When and where to use RxJS

Learning to use a new tool is as important as learning when not to use it. The types of data sources we'll be dealing with in this book can be classified into the four different categories listed in figure 2.10, which we'll explain next.

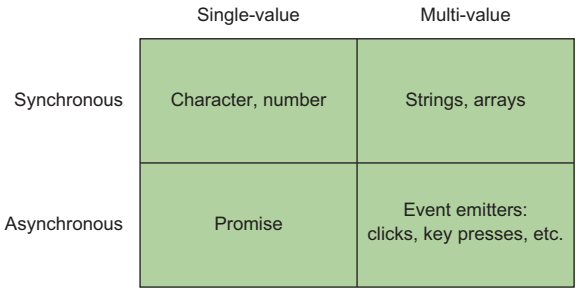


Figure 2.10 Different types of data sources with examples in each quadrant

SINGLE-VALUE, SYNCHRONOUS

The simplest case is that you have only a single piece of data. In programming, you know there are operations that return a single value for each invocation. This is the category of any function that returns a single object. You can use the `Rx.Observable.of()` function to wrap a single, synchronous value. As soon as the subscriber is attached, the value is emitted (we haven't yet explained the details behind `subscribe`, but we'll cover that in a bit):

```
Rx.Observable.of(42).subscribe(console.log); //-> 42
```

Although there are cases where you'll need to wrap single values, in most cases, if your goal is just to perform simple operations on them (concatenating another string, adding another number, and others), an observable wrapper may be overkill. The only time you'll wrap simple values with observables is when they combine with other streams.

MULTI-VALUE, SYNCHRONOUS

You can also group single items together to form collections of data, mainly for arrays. In order to apply the same operation that you used on the single item on all of the items, you would traditionally iterate over the collection and repeatedly apply the same operation to each item in the collection. With RxJS, it works in exactly the same way:

```
Rx.Observable.from([1, 2, 3]).subscribe(console.log);
// -> 1
      2
      3

Rx.Observable.from('RxJS').subscribe(console.log);
// -> "R"
      "x"
      "J"
      "S"
```

The RxJS `from()` operator is probably one of the most commonly used. And to make it a bit more idiomatic, RxJS has overloaded the `forEach` observable method as well, with the exact same semantics as `subscribe`:

```
const map = new Map();

map.set('key1', 'value1');
map.set('key2', 'value2');

Rx.Observable.from(map).forEach(console.log);
//-> ["key1", "value1"] ["key2", "value2"]
```

Both of these groups operate synchronously, which means each subsequent block of code must wait for the previous block to complete before executing. In the multi-value example, each item will be processed serially (one by one) until the collection is exhausted. This behavior is useful when dealing with items that have been preallocated, like arrays, sets, or maps, or if they can be generated, in place, on demand. Essentially, you can consider synchronous behavior to be actions on demand with results returning immediately (or at the very least before any further processing is done). When this is not the case, data is known as asynchronous.

SINGLE-VALUE, ASYNCHRONOUS

This brings us to the second dimension of computing tasks, where RxJS gives you the most benefits. This dimension addresses whether a task will execute synchronously or asynchronously. In the latter case, code is only guaranteed to run at some time in the future; thus, subsequent code blocks can't rely on any execution of a previous block having already taken place. Like with the first dimension, you also have a single-value case, where the result of a task will result in a single return value. This kind of operation is usually used to load some remote resource via an AJAX call or wait on the result of some non-local calculation wrapped in a `Promise`, without blocking the application. In either case, after the operation is initiated, it will expect a single return value or an error.

As we mentioned previously, in JavaScript this case is often handled using `Promises`. A `Promise` is similar to the single-value data case in that it resolves or errors only once. RxJS has methods to seamlessly integrate with `Promises`. Consider this simple example of a `Promise` resolving into a single, asynchronous value:

```
const fortyTwo = new Promise((resolve, reject) => {
   setTimeout(() => {
      resolve(42);
   }, 5000);
});

Rx.Observable.fromPromise(fortyTwo)
   .map(increment)
   .subscribe(console.log); //-> 43

console.log('Program terminated');
```

NOTE The promised value is being computed asynchronously, but `Promises` differ from `Observables` in that they're executed *eagerly*, as soon as they're declared.

Running this program as is produces the following output:

```
'Program terminated'

43 //-> after 5 seconds elapse
```

And because `Promises` are single-value and immutable, they're never run again. So if you subscribe to one 10 seconds later, it will return the same value 10 times—this is a desirable trait of a `Promise` by design. In chapter 7, you'll learn that you can retry a `Promise Observable` and force it to be executed many times by nesting it within another `Observable`, which has support for retries. Using the version of `ajax(url)` that returns a `Promise`, you can write the following:

```
Rx.Observable.fromPromise(ajax('/data'))
   .subscribe(data => console.log(data.id));
```

Another frequently used alternative is to use jQuery's deferred objects, which also implement the `Promise` interface. In particular, you can use functions like `$.get(url)` or `$.getJSON(url)`:

```
Rx.Observable.fromPromise($.get('/data'))
   .subscribe(data => console.log(data.id));
```

MULTI-VALUE, ASYNCHRONOUS

For those keeping score, this brings us to our fourth and final group of computing tasks. The tasks in the fourth group are those that will produce multiple values over time, yet do so asynchronously. You create this category especially for the DOM events, which are all asynchronous and can occur infinitely many times. This means that you'll need a mix of semantics from both the iterator and the promise patterns. More specifically, you need a way to process infinitely many items in sequence and capture any errors that occur. These items could be data fetched from remote AJAX calls or data generated from dragging the mouse across the screen. For this you need to invert your control structures to operate asynchronously.

The typical solution to a problem of this nature would be to use an `EventEmitter`. It provides hooks or callbacks to which closures can be passed; in this way it's very much like the `Promise`. But an event emitter doesn't stop after a single event; instead, it can continue to invoke the registered callbacks for each event that arrives, creating a practically infinite stream of events. The emitter will fulfill both of your criteria for handling multi-value, asynchronous events. But it's not without its share of problems. Though simple to use, event emitters don't scale well for larger systems, because their simplicity leads to a lack of expressiveness. The semantics for unsubscribing and disposing of them can be cumbersome, and there's no native support for error handling. These deficits can make it difficult to compose and synchronize complex tasks where multiple events from different parts of the system can be in flight simultaneously.

Rather, you can use RxJS to wrap event emitters, with all their benefits and versatility. The following code attaches a callback to a `click` event on a `link` HTML element:

```
const link = document.querySelector('#google');

const clickStream = Rx.Observable.fromEvent(link, 'click')
  .map(event => event.currentTarget.getAttribute('href'))
  .subscribe(console.log); //-> http://www.google.com
```

Queries the DOM for the link HTML element

Creates an observable around click events on this link

Extracts the link's href attribute

Note that in this example, the `subscribe()` method was used to process `click` events and perform the required business logic, in this case extracting the `href` attribute, as shown in figure 2.11. Later on, when we cover the `Observable` instance methods that form the pipeline, you'll see concrete examples of how to decouple the business logic from the printing of the result.

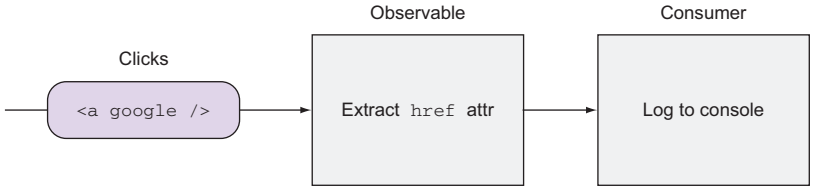


Figure 2.11 Observable that wraps `click` events and passes them down to the observer for processing

You can also use `Observables` to wrap any custom event emitters. Going back to our calculator emitter in Node.js, instead of listening for the `add` event,

```
addEmitter.on('add', (a, b) => {
  console.log(a + b); //-> Prints 5
});
```

you can subscribe to it:

```
Rx.Observable.fromEvent(addEmitter, 'add', (a, b) => ({a: a, b: b}))
   .map(input -> input.a + input.b)
   .subscribe(console.log); //-> 5

addEmitter.emit('add', 2, 3);
```

In this section, we covered only a few of the ways for creating `Observables` with RxJS. Later on, we'll tackle more-complex problems as well as new `Observable` methods.

2.3.4 To push or not to push

Event emitters have been around as long as the JavaScript language. In that time, they haven't had any significant improvements to their interface in the latest releases of the language. This contrasts with `Promises`, iterators, and generators, which were part of the JavaScript ES6 specification and are already supported in many browsers at the time of writing. This is one of the reasons why RxJS is so important; it brings many improvements to JavaScript's event system.

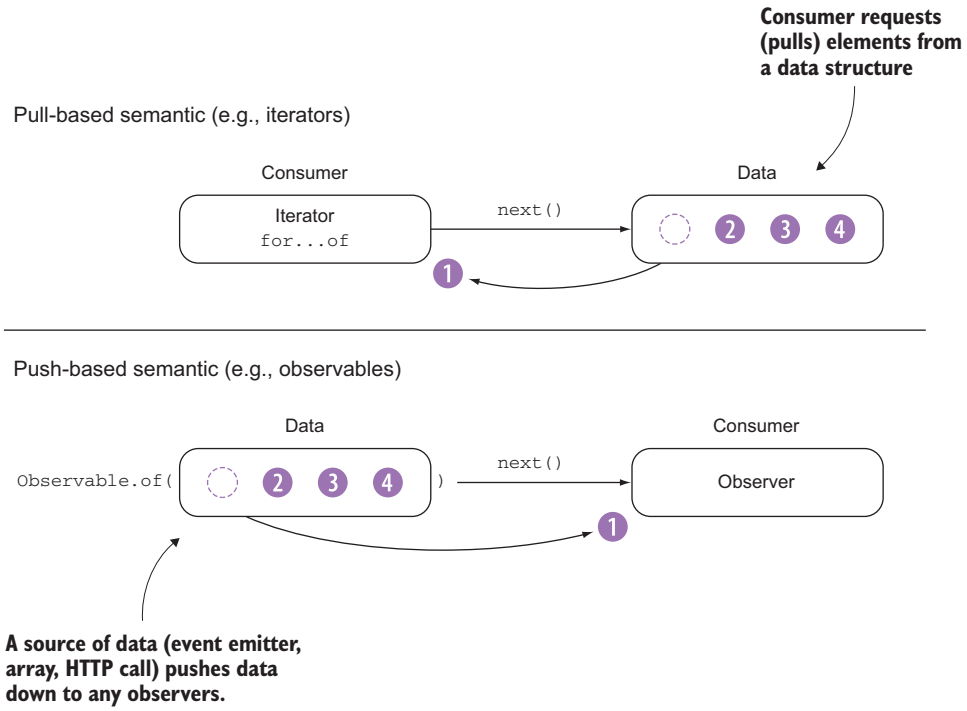


Figure 2.12 **Notice the positions of the consumer and the direction of the data. In pull-based semantics, the consumer requests data (iterators work this way), whereas in pushed-based semantics, data is sent from the source to the consumer without it requesting it. Observables work this way.**

Event emitters parse through a sequence of events asynchronously, so they come really close to being an iterator and, hence, a stream. The difference, however, lies in the way data is consumed by its clients—whether it is pulled or pushed. This is extremely important to understand, because most of the literature for RxJS defines observables as objects that represent *push-based* collections. Figure 2.12 highlights the main difference between the pull and push mechanisms, which we'll explain immediately.

Iterators use a pull-based semantic. This means that the consumer of the iterator is responsible for requesting the next item from the iterator. This data-on-demand model has two major benefits. First, it creates an abstraction over the data structure that's being used. Essentially, any data source that exposes some common method of iteration can be used interchangeably with another. The second benefit of data on demand is for sequences of data that result from some calculation. Such is the case with JavaScript generators.

For instance, for a Fibonacci number sequence, which is infinite, you need only calculate numbers as they're requested rather than wasting computing time generating parts of a sequence that the caller doesn't care about. This is immensely helpful if

the data source is expensive or difficult to calculate. In the next listing, you use a generator to create a lazy Fibonacci calculator. Generators are nothing more than iterators behind the scenes, so each value will be produced only when the consumer calls (or pulls) the `next()` method.

Listing 2.3 Fibonacci function using generators

```
function* fibonacci() {
  let first = 1, second = 1;
  for(;;) {
    let sum = second + first;
    yield sum;
    first = second;
    second = sum;
  }
}

const iter = fibonacci();

console.log(iter.next()); //-> {value: 2, done: false}
console.log(iter.next()); //-> {value: 3, done: false}
console.log(iter.next()); //-> {value: 5, done: false}
```

A generator function is denoted by the * (star) notation.

Fibonacci sequence must be initialized with at least two values.

yield will return the result of each intermediate step in the loop.

Creates the generator

Want to learn more about generators?

Generators are a language feature added into JavaScript as part of the ES6 specification. From a syntax point of view, generators introduce the `function*` and `yield` keywords. A function with an asterisk declares that a function behaves as a generator, which means it can exit with a return value via `yield` and later reenter. Under the hood, generators don't actually execute immediately but return an `Iterator` object, which is accessed via its `next()` method. Through this `Iterator` object, a generator can pause and resume exactly where it left off, and any context (closure) is kept across reentrances. A generator is a rare but powerful construct for producing infinite data using a given formula or template. If you want to learn more about them, we recommend you read the documentation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*.

A pull-based paradigm is useful in cases where you know that a value can be returned immediately from a computation. But in scenarios like listening for a mouse click, where the consumer has no way of knowing when the next piece of data will become available, this paradigm breaks down. For this reason, you require a corresponding type on the asynchronous side that is *push-based*—the opposite of the pull-based approach. In a push paradigm, the producer is responsible for creating the next item, whereas the consumer only listens for new events. As an example of this, consider your phone's email client. A pull-based mechanism that checks for new email every second can drain the resources of your mobile device quickly, whereas with push email, or any push notifications for that matter, your email client needs to react to any incoming messages only once.

RxJS observables use push-based notifications, which means they don't request data; rather, data is pushed onto them so that they can react to it. Push notifications bring the reactive paradigm to life. RxJS proposes observables as an improvement over event emitters because they're more versatile and extensible. The observable also serves as a better contemporary to the `Iterator` type, given that it possesses similar semantics but with a push-based mechanism.

You can see from our discussion so far how iterators and `Promises` can be potential data sources that can be wrapped as observables, even though we earlier classified them as distinct groups. This ability to adapt not just the types they are replacing but also types from other groups is immensely powerful—observables work equally well across synchronous and asynchronous boundaries. It not only makes interfacing with legacy code incredibly easy, but also it allows consumer code to be written independently of how the producer is implemented.

> **WATCH OUT!** This power comes with responsibility as well, for although you're *able* to convert anything your heart desires into `Observables`, it doesn't always mean that you *should*. In particular, processes that are strictly synchronous and iterative or will only ever deal with a single value do not need to be "Rx-ified" just for the sake of being cool. Even though `Observables` are cheap to create, there's a bit of overhead associated with applying simple operations on data. For instance, just transforming a string from lower- to uppercase does not require it to be wrapped with an observable; you should directly use the string methods. Don't be reactive just because you can.

In RxJS, you'll always have a pipeline that takes data from the source to the corresponding consumer. Data will always be created or materialized from a data source. Again, the type of data source isn't relevant to how your abstraction operates; when data reaches the end of its journey and must be consumed, it's immaterial where the data came from. We'll reiterate that the separation and abstraction of these two concepts, data production and data consumption, is important for three reasons:

- It enables you to hide differences of implementation behind a common interface, which lets you focus more on the business logic of your task. This has the benefit of not only optimizing development time but also reducing code complexity by removing extra noise from code.
- The separation of production and consumption builds a clear separation of concerns and makes the direction of data flow clear.
- It makes streams testable by allowing you to attach mock versions of the producer and wire the corresponding matching expectations in the observer.

Now that you understand how streams can be constructed, you're missing only the last place where observers come into play—stream consumption.

2.4 Consuming data with observers

Every piece of data that's emitted and processed through an observable needs a destination. In other words, what was the purpose of capturing and processing a certain event? Observers are created within the context of a subscription, which means that the result of calling `subscribe()` on an observable source is a `Subscription` object. Because observables operate synchronously or asynchronously, the consumer of an observable must in some way support the inversion of control that also happens with callbacks. This is consistent with its push-based mechanism. That is, because you don't know when a DOM element, for instance, will fire an event or when the result of an AJAX call will return, observables must be able to call into or signal the observer structure that more data is available by using the observer's `next()` method, as illustrated in figure 2.13. This mechanism is directly inspired in the iterator and observer patterns. An iterator doesn't know (or care) about the size of the data structure it's looping over or if it will ever end; it only knows whether there's more data to process.

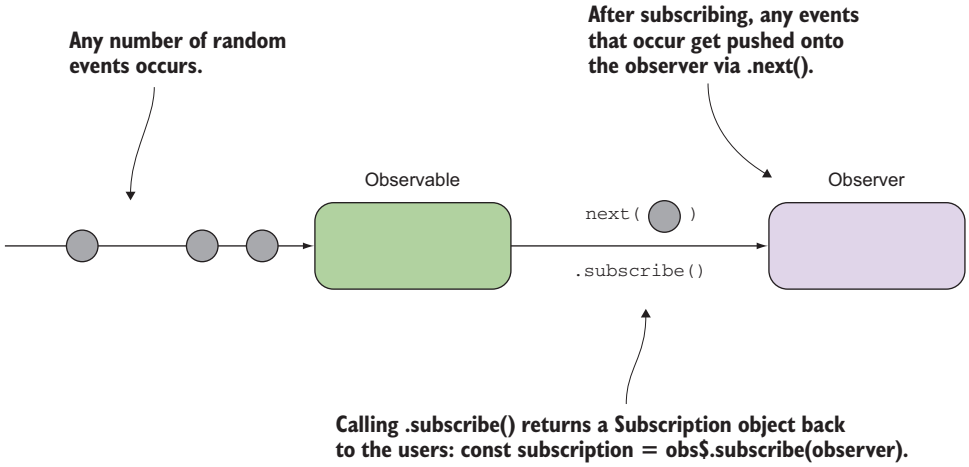


Figure 2.13 Observables calling into an observer's methods. Observers expose a simple iterator-like API with a next() method. Upon subscription, an object of type `Subscription` is returned to the calling code, which it can use for cancellation and disposal, as we'll discuss in a bit.

Through a concise iterator-like API, observables are able to signal to their subscribers whether more events have occurred. This gives you the flexibility to control what data observers receive.

2.4.1 The Observer API

An observer is registered with an observable in much the same way that you registered callbacks on an event emitter. An observable becomes aware of an observer during the subscription process, which you've seen a lot of so far. The subscription process is a way for you to pass an observer reference into an observable, creating a managed, one-way relationship.

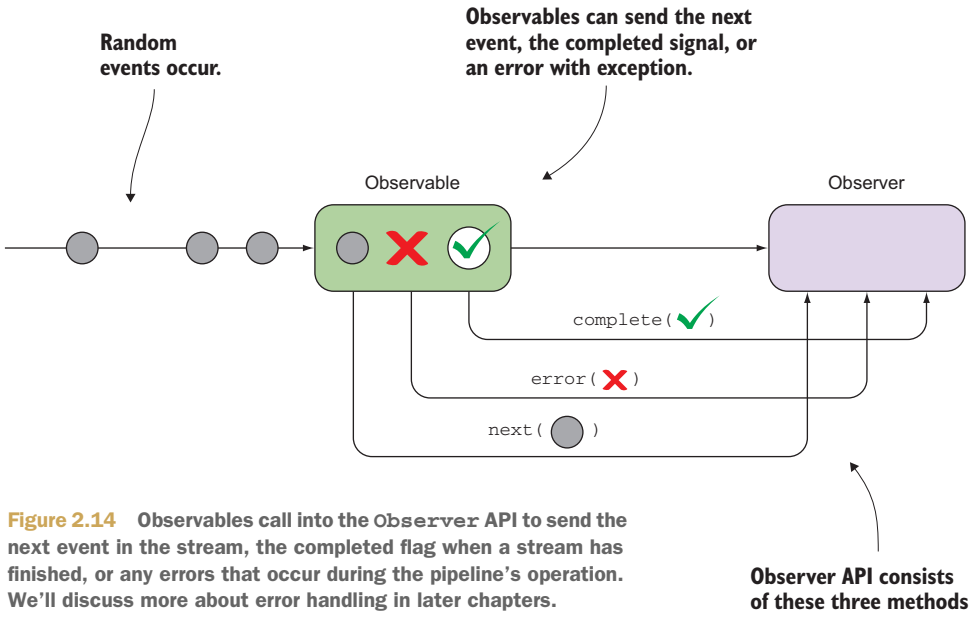


Figure 2.14 Observables call into the `Observer` API to send the next event in the stream, the completed flag when a stream has finished, or any errors that occur during the pipeline's operation. We'll discuss more about error handling in later chapters.

Figure 2.14 shows how observables call an observer's methods to signal more data, completion, and even errors. As you can see, aside from `next()`, two other methods are called on observers: `error()` and `complete()`.

Figure 2.14 shows that once the `subscribe` method is called, an observer is implicitly created with an API that exposes three (optional) methods: `next`, `complete`, and `error` (in RxJS 4 these were called `onNext`, `onCompleted`, and `onError`, respectively). In code, the resulting object has the following structure:

```
const observer = {
    next: function () {
        // process next value
    },
    error: function () {
        // alert user
    },
    complete: function () {

    }
}
```

Up until now, you've used a single function call only to process the results. This function maps to `next()`. Each method serves a specific purpose in the lifetime of the observer, as shown in table 2.1.

Alternatively, you can use this API directly by creating your own observable.

Table 2.1 Defining the `Observer` API

Name	Description
`next(val):void`	Receives the next value from an upstream observable. This is the equivalent of `update` in the observer pattern. When a single function is passed into `subscribe()` instead of an observer object, it maps to the observer's `next()`.
`complete():void`	Receives a completion notification from the upstream observable. Subsequent calls to `next()`, if any, are ignored.
`error(exception):void`	Receives an error notification from the upstream observable. This indicates that it encountered an exception and won't be emitting any more messages to the observer (subsequent calls to `next()` are ignored). Generally, error objects are passed in, but you could customize this to pass other types as well.

2.4.2 Creating bare observables

Most of the time, you'll use the RxJS factory operators like `from()` and `of()`, as you learned at the beginning of this chapter, to instantiate observables. In practice, these should cover all your needs. But it's important to understand how observables work under the nice RxJS abstraction and how they interact with the observer to emit events. We'll show you a barebones model of an observable that emits events asynchronously and exposes the mechanism to unsubscribe. At the core, an observable is a function that processes a set of inputs and returns a subscription to the caller to manage the disposal of the stream:

```
const observable = events => {
  const INTERVAL = 1 * 1000;
  let schedulerId;

  return {
     subscribe: observer => {
       schedulerId = setInterval(() => {
         if(events.length === 0) {
            observer.complete();
            clearInterval(schedulerId);
            schedulerId = undefined;
         }
         else {
            observer.next(events.shift());
         }
       }, INTERVAL);

       return {
          unsubscribe: () => {
             if(schedulerId) {
                 clearInterval(schedulerId);
             }
          }
       };
```

```
      }
    }
};
```

You can call this function by passing the observer object:

```
let sub = observable([1, 2, 3]).subscribe({
  next: console.log,
  complete: () => console.log('Done!')
});
//-> 1
       (...1 second)
     2
       (...1 second)
     3
       (...1 second)
     Done!
```

This is a simplistic model of RxJS, and there's much more that goes into it. But the main takeaway here is that an observable behaves like a function that begins chipping away at the data pushed into it as soon as a subscriber is available; the subscriber has the key to turn the stream off via `sub.unsubscribe()`. Now, let's move on to using RxJS.

Using RxJS, you can register an observer object through `Rx.Observable.create()`. Like the previous code, this function expects an observer object that you can use to signal the next emitted event by invoking its `next()` method. Most of the time, you'll provide the observer object literal directly into the subscription and use the static `create()` method when you want full control of how and when the data is emitted from the observable through the `Observer` API. For instance, you create observables artificially by calling into the observer's methods directly:

```
const source$ = Rx.Observable.create(observer => {
   observer.next('4111111111111111');
   observer.next('5105105105105100');
   observer.next('4342561111111118');
   observer.next('6500000000000002');
   observer.complete();
});

const subscription = source$.subscribe(console.log);
```

If an observable is finite, you can signal its completion by calling the observer's complete() method.

At this point, the observable stands idle and none of the data is emitted or passed into the observer.

With subscribe(), the observer logic is executed; in this case, it's printing to the console.

A marble diagram of this stream would look like figure 2.15.

This sample code is simple because it just emits a series of account numbers, but you could do much more. You could create your own observables with custom behavior that can be reused anywhere in your application.

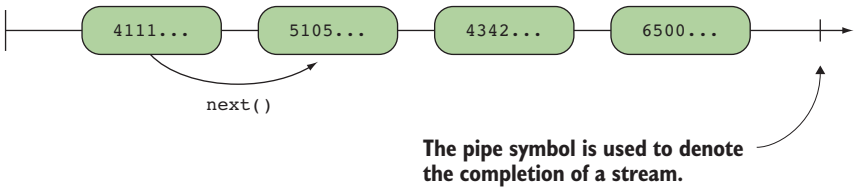


Figure 2.15 A marble diagram showing a synchronous set of events ended by a call to complete()

2.4.3 Observable modules

Directly calling the observer object allows you to define the data that's pushed to the subscriber. How this data is generated and where it comes are encapsulated into the observable's context—kind of like a module. For instance, suppose you wanted to create a simple progress indicator widget that can be used when a user is performing a long-running operation. This module will emit percentage values 0% to 100% at a certain speed, as shown in the following listing.

Listing 2.4 Custom progress indicator module using RxJS

```
const progressBar$ = Rx.Observable.create(observer => {
   const OFFSET = 3000;
   const SPEED =  50;

   let val = 0;
   function progress() {
     if(++val <= 100) {
       observer.next(val);
       setTimeout(progress, SPEED);
     }
     else {
       observer.complete();
     }
   };
   setTimeout(progress, OFFSET);
});

const label = document.querySelector('#progress-indicator');

progressBar$
  .subscribe(
    val   => label.textContent = (Number.isInteger(val) ? val + "%" : val),
    error => console.log(error.message),
    ()    => label.textContent = 'Complete!'
);
```

Emits a new progress value every 50 milliseconds

Starts the progress indicator counter after three seconds

Calls the progress function recursively

Sends the complete signal after reaching 100%

The business logic of how the values are generated and emitted belongs in the observable, whereas all the details of rendering, whether you want a simple number indicator or use some third-party progress bar widget, are for the caller to implement within the observer.

NOTE You could also achieve this by using RxJS's time operators. More about this in the next chapter.

Using these methods gives you more opportunities to react to the different states of the program. Stepping back into our discussion about iterators and generators in chapter 2, observers operate similarly to these artifacts. The key difference is that the iterator uses a pull-based mechanism as opposed to an observable's push-based nature—an observable pushes values into an observer. For iterators and generators, the consuming code is controlling the pace of consumption. For instance, a `for` loop controls (or requests) what to pull from an iterator or a generator, not the other way around. This means that each time a new piece of data is needed (by a call to `next()` or `yield`), the consumer of the iterator will call the appropriate method to advance the state of the iterator. Figure 2.16 shows another example using the Fibonacci sequence.

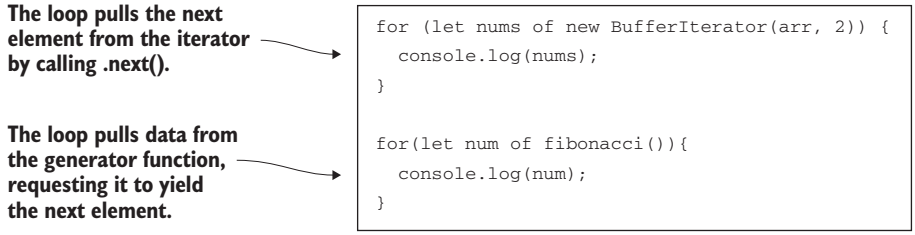


Figure 2.16 **The pull mechanism of iterators**

As a result, iterators must have a way to inform the consumer that there are no longer any items for consumption. Bank tellers are real-world iterators. Each time a customer comes up, that person must be handled before the next customer can be helped. When the teller becomes available, they yell "Next!" to "pull" the next customer in. If they were to call "Next!" and no one responded, they would know that the line was complete and it might be safe to take their lunch break.

Something to keep in mind, though, is that infinite event emitters, like the DOM, will never fire the `complete()` function (or `error()` for that matter) on any of its events. Therefore, it's entirely up to you to unsubscribe from them or roll your own autodispose mechanism. But for finite event sequences, when an observer is called with either of these methods, it knows that contractually it won't receive any more messages from its owning observable. This again is a tight parallel to an iterator, which by definition should stop returning values when the iteration generates an exception or completes.

Consider a simple `Promise` object that resolves to the value `42` after 5 seconds (shown in figure 2.17).

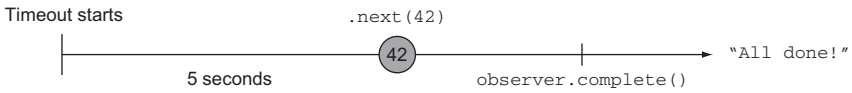


Figure 2.17 An observable (wrapped `Promise`) that emits a value after 5 seconds

We mentioned in chapter 1 that `Promises` can be used to model an immutable, single (future) value. You'll use the `setTimeout()` function to simulate this; now, instead of creating your own observable, you'll use the generic creational methods in RxJS, such as the following:

```
Rx.Observable.fromPromise():
const computeFutureValue = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(42);
  }, 5000);
});
```

Resolves the `Promise` after 5 seconds have elapsed

```
Rx.Observable.fromPromise(computeFutureValue)
  .subscribe(
    val => {
      console.log(val);
    },
    err => {
      console.log(`Error occurred: ${err}`);
    },
    () => {
      console.log('All done!');
    });
```

Because `Promises` emit a single value, this stream will eventually send the completed status after 5 seconds have passed, printing "All done!" at the end. Now, suppose that instead of a resolved `Promise`, something goes wrong in computing this value and the `Promise` is rejected:

```
const computeFutureValue = new Promise((resolve, reject) => {
  setInterval(() => {
    reject(new Error('Unexpected Exception!'));
  }, 5000);
});
```

This will cause the observable to invoke the `error()` method on the observer and print the following message after 5 seconds:

```
"Error occurred: Unexpected Exception!"
```

This is quite remarkable because RxJS not only takes care of error handling for you (without messy, imperative `try/catch` statements) but also provides logic that ties in with `Promise` semantics of resolve/reject. We'll cover all there is to know about error handling in chapter 7.

An important takeaway from this discussion about observers is that the callbacks passed to it are, for all practical purposes, future code. That is, you don't know when the callbacks will actually be called, so other code shouldn't make assumptions about their execution. This relates to the larger point made earlier about the nature of the code within a stream. Because one of your goals is to move away from the messy business of keeping track of state changes, avoiding the introduction of side effects is one of the ways that you can keep your streams pure and prevent unwanted changes from adversely seeping into the application logic. This works well with RxJS because pure functions can run in any order and at any time (now or in the future) and will always yield the correct results.

With observers, we've finish introducing the three main parts of RxJS: producers (observables), the pipeline (business logic), and consumers (observers). This chapter is just the start of your journey of learning how to think reactively (and functionally). It will take much more time and many more examples to truly understand how you can think reactively, but you were able to get your feet wet on some advanced APIs. Much of what you've seen so far has been abstract in nature with very little coding, but this step is crucial for understanding how this approach differs from ones you've been taught in the past. In the next chapter, we'll look more closely at the operations that you can perform on streams as well as how you can cancel them if needed. By doing so, we're officially taking the training wheels off and introducing you to the core operations for building applications in RxJS.

2.5 Summary

- RxJS and, more generally, the concept of thinking in streams derive many of their foundational principles from functional programming.
- The declarative style of RxJS allows you to translate almost exactly from your problem statement into working code.
- Data sources can often operate quite differently, even within the observable contract.
- Mouse clicks, HTTP requests, or simple arrays are all the same under the eyes of observables.
- Push-based and pull-based semantics are represented through observables and iterators, respectively. Wrapping data sources is the first step in creating a pipeline/observable.
- Observables abstract the notion of production and consumption of events such that you can separate production, consumption, and processing into completely self-contained constructs.
- Observers expose an API with three methods: `next()`, `complete()`, and `error()`.

In the most general sense, React is a JavaScript library for building user interfaces across a variety of platforms.In this chapter, we'll examine where React fits in the broader world of web engineering and what makes React's mental model powerful. We'll also get a high-level view of React components, the virtual DOM, and some of React's concepts and paradigms.

Chapter 1 from *React in Action*
by Mark Tielens Thomas

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

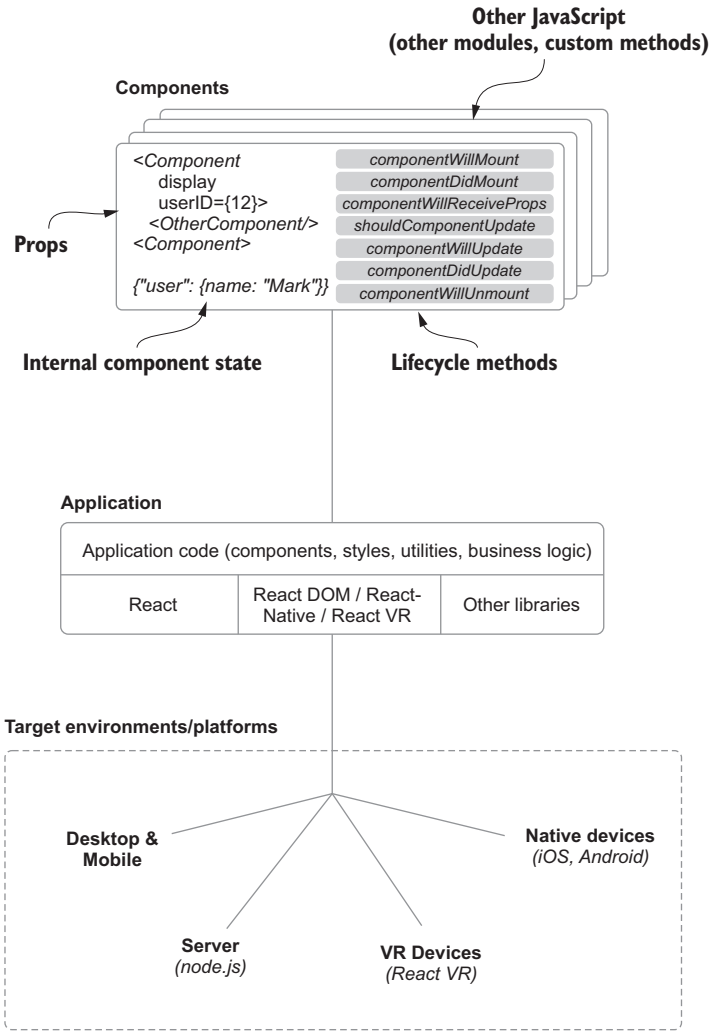


Figure 1.1 React allows you to create user interfaces from components. Components maintain their own state, are written in and work with "vanilla" JavaScript, and inherit a number of helpful APIs from React. Most React apps are written for browser-based environments, but can also be used in native environments 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.

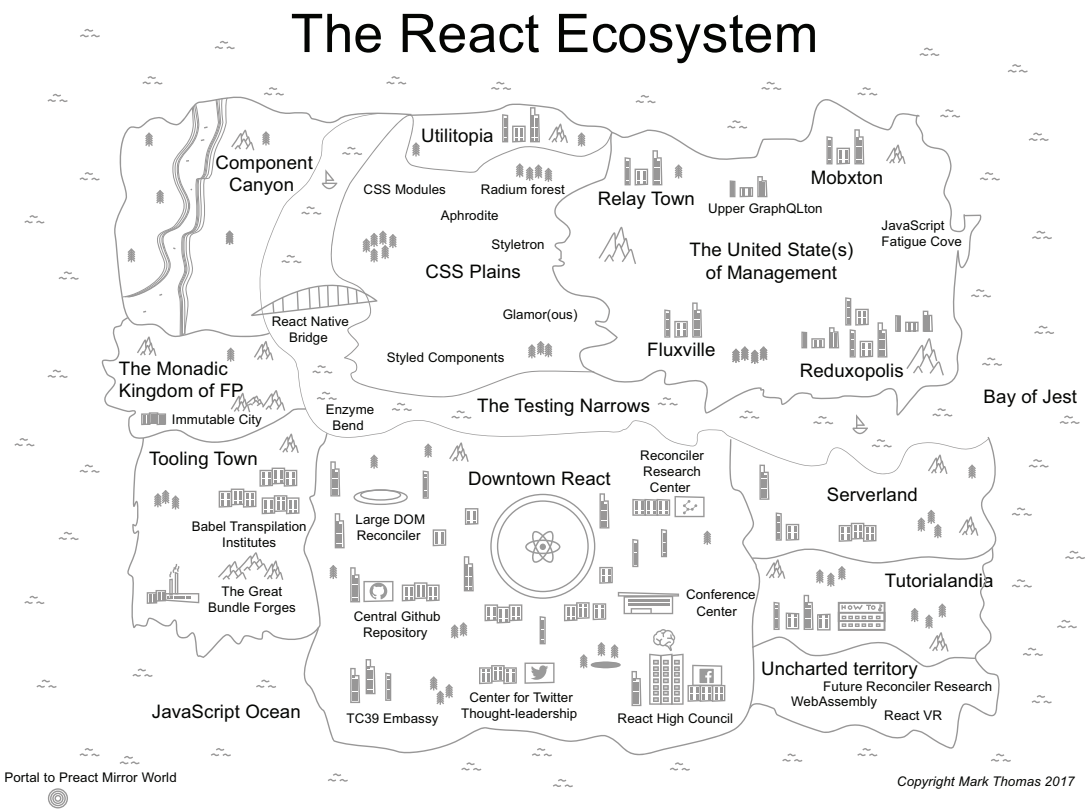


Figure 1.2 A map of the React ecosystem is diverse—even more so than I can represent here. If you'd like to learn more, check out my guide at https://ifelse.io/react-ecosystem, which will help you find your way in the React ecosystem when starting out.

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.

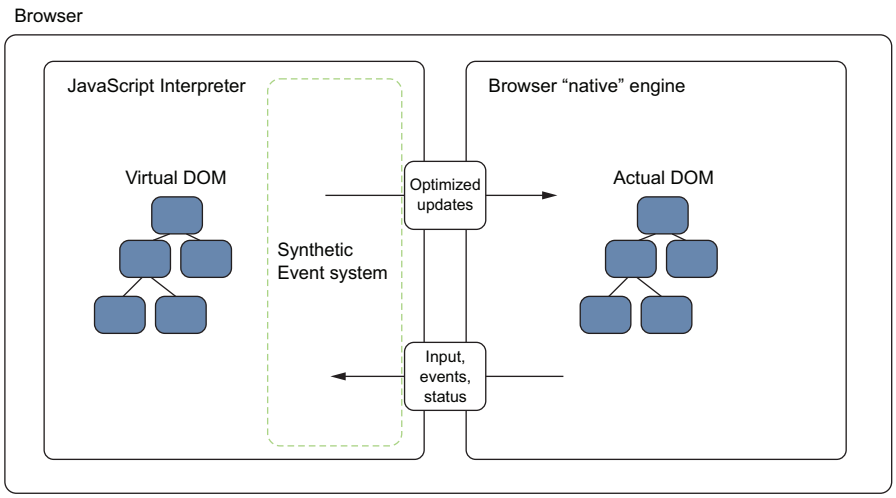


Figure 1.3 The DOM and virtual DOM. React's virtual DOM handles change detection in data as well as translating browser events into events that React components can understand and react to. React's virtual DOM also aims to optimize changes made to the DOM for the sake of performance.

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.

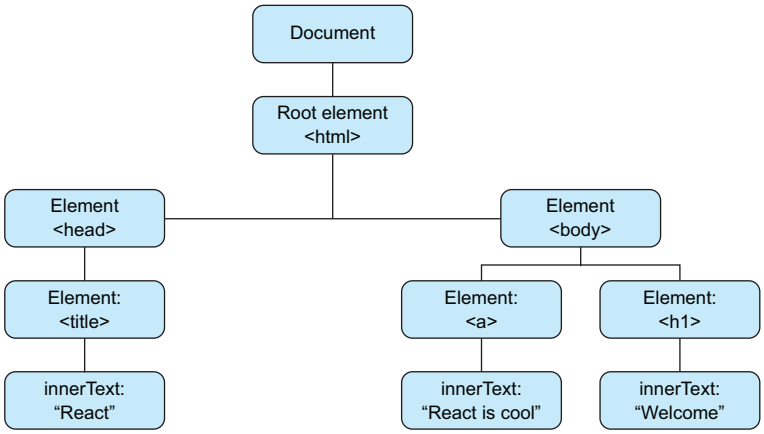


Figure 1.4 Here's a simple version of the DOM tree structure, using elements you're probably familiar with. The DOM API that's exposed to JavaScript lets you performs operations on these elements in the tree.

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.

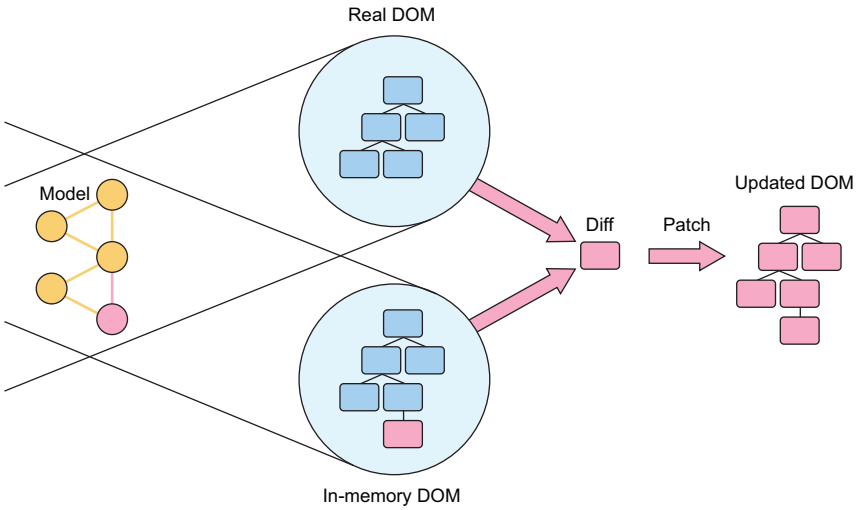


Figure 1.5 React's diffing and update procedure. When a change happens, React determines differences between the actual and in-memory DOMs. Then it performs an efficient update to the browser's DOM. This process is often referred to as a *diff* ("what changed?") *and patch* ("update only what changed")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

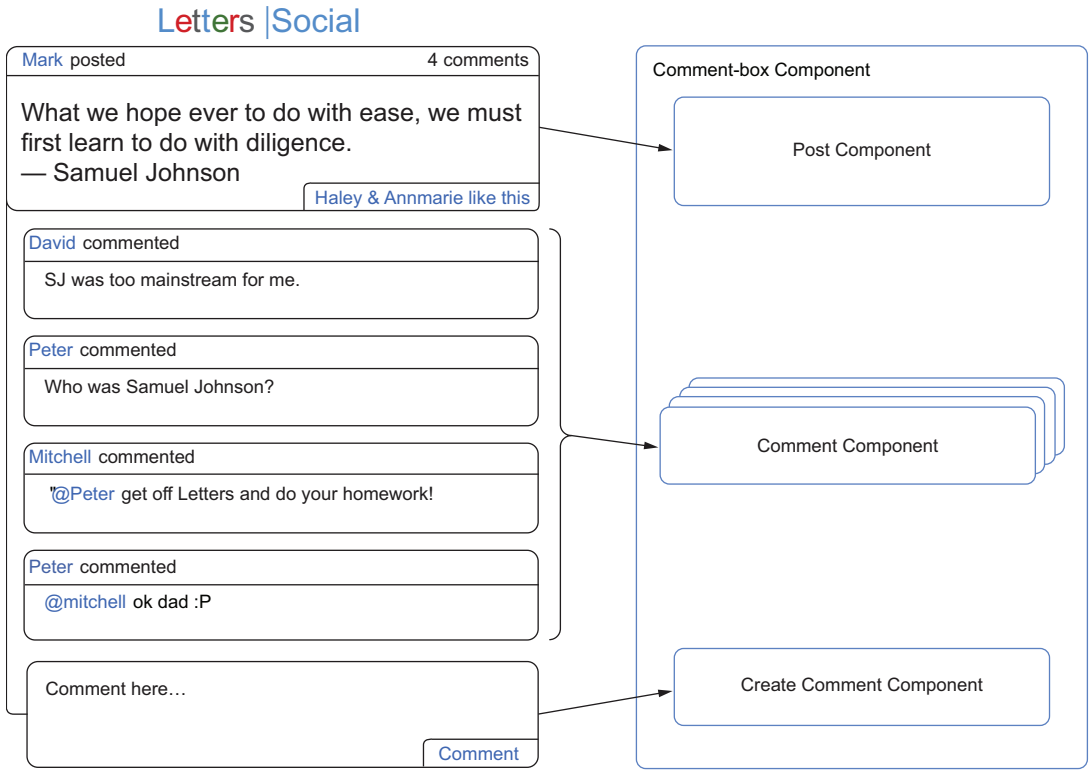


Figure 1.6 **An example of an interface broken into components. Each distinct section can be thought of as a component. Items that repeat in a uniform nature can be thought of as one component that gets reused over different data.**

> **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.

In this chapter, you'll get the lowdown on what OAuth 2.0 is, what it does, and why it's important.

Chapter 1 from *OAuth 2 in Action* by Justin Richer and Antonio Sanso

What is OAuth 2.0 and why should you care?

This chapter covers

- What OAuth 2.0 is
- What developers do without OAuth
- How OAuth works
- What OAuth 2.0 is not

If you're a software developer on the web today, chances are you've heard of OAuth. It is a security protocol used to protect a large (and growing) number of web APIs all over the world, from large-scale providers such as Facebook and Google to small one-off APIs at startups and inside enterprises of all sizes. It's used to connect websites to one another and it powers native and mobile applications connecting to cloud services. It's being used as the security layer for a growing number of standard protocols in a variety of domains, from healthcare to identity, from energy to the social web. OAuth is far and away the dominant security method on the web today, and its ubiquity has leveled the playing field for developers wanting to secure their applications.

But what is it, how does it work, and why do we need it?

1.1 What is OAuth 2.0?

OAuth 2.0 is a delegation protocol, a means of letting someone who controls a resource allow a software application to access that resource on their behalf without impersonating them. The application requests authorization from the owner of the resource and receives *tokens* that it can use to access the resource. This all happens without the application needing to impersonate the person who controls the resource, since the token explicitly represents a delegated right of access. In many ways, you can think of the OAuth token as a "valet key" for the web. Not all cars have a valet key, but for those that do, the valet key provides additional security beyond simply handing over the regular key. The valet key of a car allows the owner of the car to give limited access to someone, the valet, without handing over full control in the form of the owner's key. Simple valet keys limit the valet to accessing the ignition and doors but not the trunk or glove box. More complex valet keys can limit the upper speed of the car and even shut the car off if it travels more than a set distance from its starting point, sending an alert to the owner. In much the same way, OAuth tokens can limit the client's access to only the actions that the resource owner has delegated.

For example, let's say that you have a cloud photo-storage service and a photo-printing service, and you want to be able to print the photos that you have stored in your storage service. Luckily, your cloud-printing service can communicate with your cloud-storage service using an API. This is great, except that the two services are run by different companies, which means that your account with the storage service has no connection to your account with the printing service. We can use OAuth to solve this problem by letting you delegate access to your photos across the different services, all without giving your password away to the photo printer.

Although OAuth is largely indifferent to what kind of resource it is protecting, it does fit nicely with today's RESTful web services, and it works well for both web and native client applications. It can be scaled from a small single-user application up to a multimillion-user internet API. It's as much at home on the untamed wilds of the web, where it grew up and is used to protect user-facing APIs of all types, as it is inside the controlled and monitored boundaries of an enterprise, where it's being used to manage access to a new generation of internal business APIs and systems.

And that's not all: if you've used mobile or web technology in the past five years, chances are even higher that you've used OAuth to delegate your authority to an application. In fact, if you've ever seen a web page like the one shown in figure 1.1, then you've used OAuth, whether you realize it or not.

In many instances, the use of the OAuth protocol is completely transparent, such as in Steam's and Spotify's desktop applications. Unless an end user is actively looking for the telltale marks of an OAuth transaction, they would never know it's being used.[1] This is a good thing, since a good security system should be nearly invisible when all is functioning properly.

[1] The good news is that by the end of this book, you should be able to pick up on all of these telltale signs yourself.

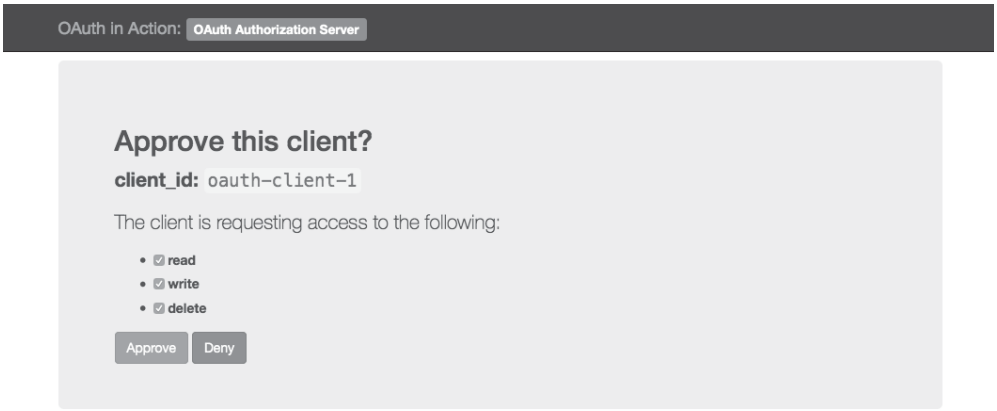


Figure 1.1 An OAuth authorization dialog from the exercise framework for this book

We know that OAuth is a security protocol, but what exactly does it do? Since you're holding a book that's purportedly about OAuth 2.0, that's a fair question. According to the specification that defines it:[2]

> *The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*

Let's unpack that a bit: as an *authorization framework*, OAuth is all about getting the right of access from one component of a system to another. In particular, in the OAuth world, a client application wants to gain access to a protected resource on behalf of a resource owner (usually an end user). These are the components that we have so far:

- The *resource owner* has access to an API and can delegate access to that API. The resource owner is usually a person and is generally assumed to have access to a web browser. Consequently, this book's diagrams represent this party as a person sitting with a web browser.
- The *protected resource* is the component that the resource owner has access to. This can take many different forms, but for the most part it's a web API of some kind. Even though the name "resource" makes it sound as though this is something to be downloaded, these APIs can allow read, write, and other operations just as well. This book's diagrams show protected resources as a rack of servers with a lock icon.
- The *client* is the piece of software that accesses the protected resource on behalf of the resource owner. If you're a web developer, the name "client" might make you think this is the web browser, but that's not how the term is used here. If you're a business application developer, you might think of the "client" as the

[2] RFC 6749 https://tools.ietf.org/html/rfc6749

person who's paying for your services, but that's not what we're talking about, either. In OAuth, the client is whatever software consumes the API that makes up the protected resource. Whenever you see "client" in this book, we're almost certainly talking about this OAuth-specific definition. This book's diagrams depict clients as a computer screen with gears. This is partially in deference to the fact that there are many different forms of client applications, as we'll see in chapter 6, so no one icon will universally suffice.

We'll cover these all in greater depth in chapter 2 when we look at "The OAuth Dance" in detail. But for now, we need to realize that we've got one goal in this whole setup: getting the client to access the protected resource for the resource owner (see figure 1.2).

In the printing example, let's say you've uploaded your vacation photos to the photo-storage site, and now you want to have them printed. The storage site's API is the resource, and the printing service is the client of that API. You, as the resource owner, need to be able to delegate part of your authority to the printer so that it can read your photos. You probably don't want the printer to be able to read all of your photos, nor do you want the printer to be able to delete photos or upload new ones of its own. Ultimately, what you're interested in is getting certain photos printed, and if you're like most users, you're not going to be thinking about the security architectures of the systems you're using to get that done.

Thankfully, because you're reading this book, chances are that you're not like most users and you do care about security architectures. In the next section, we'll see how this problem could be solved imperfectly without OAuth, and then we'll look at how OAuth can solve it in a better way.

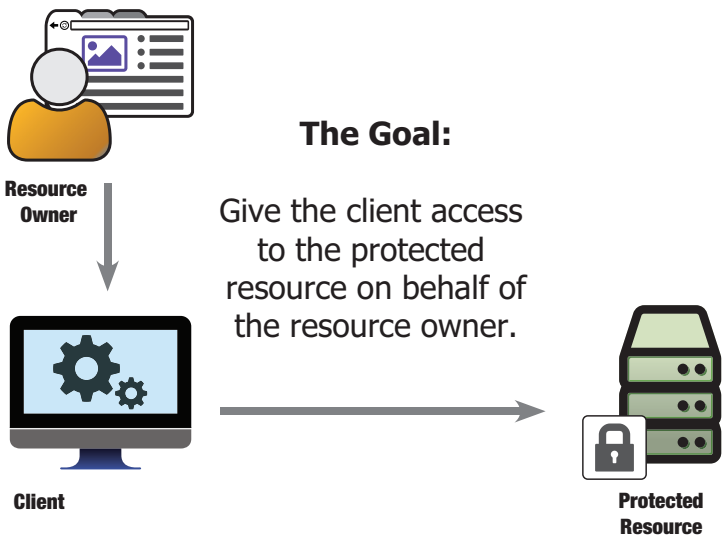


Figure 1.2 Connecting the client on behalf of the resource owner

1.2 The bad old days: credential sharing (and credential theft)

The problem of wanting to connect multiple disparate services is hardly new, and we could make a compelling argument that it's been around from the moment there was more than one network-connected service in the world.

One approach, popular in the enterprise space, is to *copy the user's credentials and replay them on another service* (see figure 1.3). In this case, the photo printer assumes that the user is using the same credentials at the printer that they're using at the storage site. When the user logs in to the printer, the printer replays the user's username and password at the storage site in order to gain access to the user's account over there, pretending to be the user.

In this scenario, the user needs to authenticate to the client using some kind of credential, usually something that's centrally controlled and agreed on by both the client and the protected resource. The client then takes that credential, such as a username and password or a domain session cookie, and replays it to the protected resource, pretending to be the user. The protected resource acts as if the user had authenticated directly, which does in fact make the connection between the client and protected resource, as required previously.

This approach requires that the user have the same credentials at the client application and the protected resource, which limits the effectiveness of this credential-theft technique to a single security domain. For instance, this could occur if a single company controls the client, authorization server, and protected resources, and all of these run inside the same policy and network control. If the printing service is offered by the same company that provided the storage service, this technique might work as the user would have the same account credentials on both services.

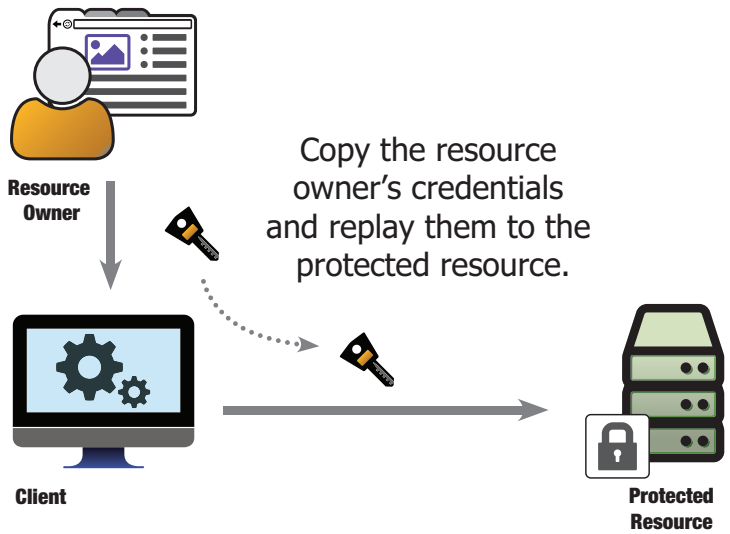


Figure 1.3 Copy the resource owner's credentials without asking

This technique also exposes the user's password to the client application, though inside a single security domain using a single set of credentials, this is likely to be happening anyway. However, the client is *impersonating* the user, and the protected resource has no way to tell the difference between the resource owner and the impersonating client because they're using the same username and password in the same way.

But what if the two services occupied different security domains, a likely scenario for our photo-printing example? We can't copy the password the user gave us to log into our application any longer, because it won't work on the remote site. Faced with this challenge, these would-be credential thieves could employ an age-old method for stealing something: *ask the user* (figure 1.4).

If the printing service wants to get the user's photos, it can prompt the user for their username and password on the photo-storage site. As it did previously, the printer replays these credentials on the protected resource and impersonates the user. In this scenario, the credentials that the user uses to log into the client can be different from those used at the protected resource. However, the client gets around this by asking the user to provide a username and password for the protected resource. *Many users will in fact do this*, especially when promised a useful service involving the protected resource. Consequently, this remains one of the most common approaches to mobile applications accessing a back end service through a user account today: the mobile application prompts the user for their credentials and then replays those credentials directly to the back end API over the network. To keep accessing the API, the client application will store the user's credentials so that they can be replayed as needed. This is an extremely dangerous practice, since the compromise of any client in use will lead to a full compromise of that user's account across all systems.

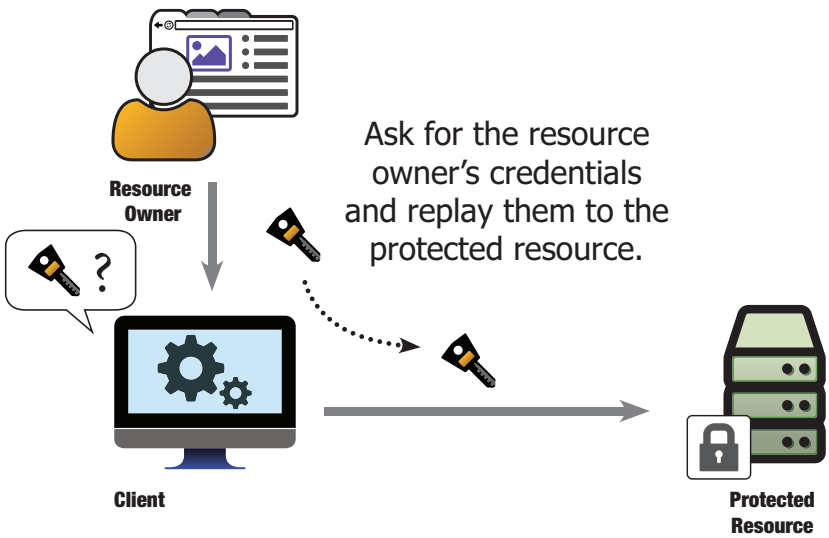


Figure 1.4 Ask for the resource owner's credentials, and replay them

This approach still works only in a limited set of circumstances: the client needs to have access to the user's credentials directly, and those credentials need to be able to be replayed against a service outside of the user's presence. This rules out a large variety of ways that the user can log in, including nearly all federated, many multifactor, and most higher-security login systems.

Lightweight Directory Access Protocol (LDAP) authentication

Interestingly, this pattern is exactly how password-vault authentication technologies such as LDAP function. When using LDAP for authentication, a client application collects credentials directly from the user and then replays these credentials to the LDAP server to see whether they're valid. The client system must have access to the plaintext password of the user during the transaction; otherwise, it has no way of verifying it with the LDAP server. In a very real sense, this method is a form of man-in-the-middle attack on the user, although one that's generally benevolent in nature.

For those situations in which it does work, it exposes the user's primary credentials to a potentially untrustworthy application, the client. To continue to act as the user, the client has to store the user's password in a replayable fashion (often in plaintext or a reversible encryption mechanism) for later use at the protected resource. If the client application is ever compromised, the attacker gains access not only to the client but also to the protected resource, as well as any other service where the end user may have used the same password.

Furthermore, in both of these approaches, the client application is *impersonating* the resource owner, and the protected resource has no way of distinguishing a call directly from the resource owner from a call being directed through a client. Why is that undesirable? Let's return to the printing service example. Many of the approaches will work, in limited circumstances, but consider that you don't want the printing service to be able to upload or delete photos from the storage service. You want the service to read only those photos you want printed. You also want it to be able to read only while you want the photos printed, and you'd like the ability to turn that access off at any time.

If the printing service needs to impersonate you to access your photos, the storage service has no way to tell whether it's the printer or you asking to do something. If the printing service surreptitiously copies your password in the background (even though it promised not to do so), it can pretend to be you and grab your photos whenever it wants. The only way to turn off the rogue printing service is to change your password at the storage service, invalidating its copy of your password in the process. Couple this with the fact that many users reuse passwords across different systems and you have yet another place where passwords can be stolen and accounts correlated with each other. Quite frankly, in solving this connection problem, we made things worse.

By now you've seen that replaying user passwords is bad. What if, instead, we gave the printing service universal access to all photos on the storage service on behalf of

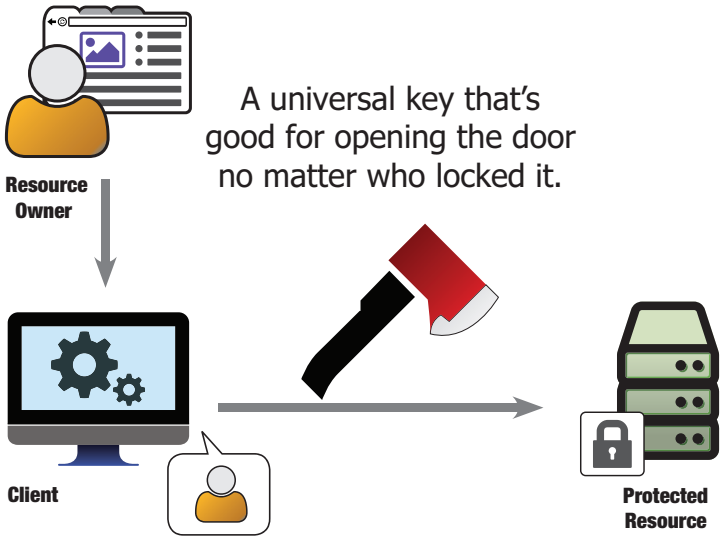


Figure 1.5 Use a universal developer key, and identify the user on whose behalf you're (allegedly) acting

anyone it chose? Another common approach is to use a developer key (figure 1.5) issued to the client, which uses this to call the protected resource directly.

In this approach, the developer key acts as a kind of universal key that allows the client to impersonate any user that it chooses, probably through an API parameter. This has the benefit of not exposing the user's credentials to the client, but at the cost of the client requiring a highly powerful credential. Our printing service could print any photos that it wanted to at any time, for any user, since the client effectively has free rein over the data on the protected resource. This can work to an extent, but only in instances in which the client can be fully known to and trusted by the protected resource. It is vanishingly unlikely that any such relationship would be built across two organizations, such as those in our photo-printing scenario. Additionally, the damage done to the protected resource if the client's credentials are stolen is potentially catastrophic, since all users of the storage service are affected by the breach whether they ever used the printer or not.

Another possible approach is to *give users a special password* (figure 1.6) that's only for sharing with third-party services. Users don't use this password to log in themselves, but paste it into applications that they want to work for them. This is starting to sound like that limited-use valet key you saw at the beginning of the chapter.

This is starting to get closer to a desirable system, as the user no longer has to share their real password with the client, nor does the protected resource need to implicitly trust the client to act properly on behalf of all users at all times. However, the usability of such a system is, on its own, not very good. This requires the user to generate, distribute, and manage these special credentials in addition to the primary passwords they already must curate. Since it's the user who must manage these credentials, there is also, generally speaking, no correlation between the client program and the credential itself. This makes it difficult to revoke access to a specific application.

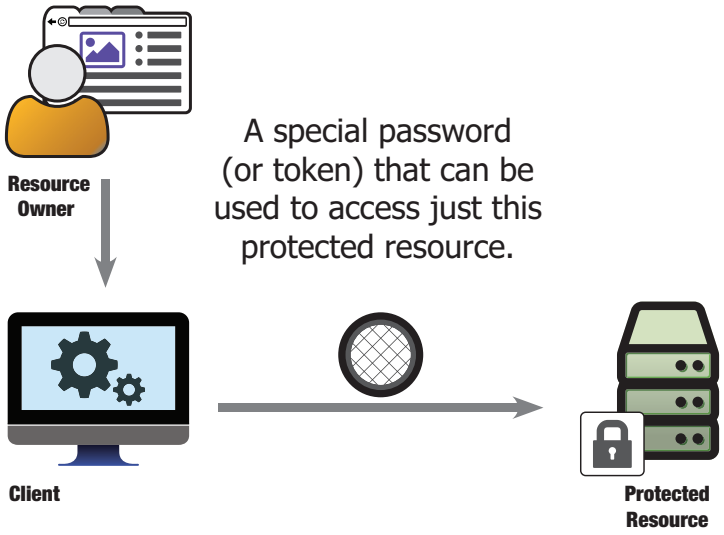


Figure 1.6 A service-specific password that limits access

Can't we do better than this?

What if we were able to have this kind of limited credential, issued separately for each client and each user combination, to be used at a protected resource? We could then tie limited rights to each of these limited credentials. What if there were a network-based protocol that allowed the generation and secure distribution of these limited credentials across security boundaries in a way that's both user-friendly and scalable to the internet as a whole? Now we're starting to talk about something interesting.

1.3 Delegating access

OAuth is a protocol designed to do exactly that: in OAuth, the end user *delegates* some part of their authority to access the protected resource to the client application to act on their behalf. To make that happen, OAuth introduces another component into the system: the *authorization server* (figure 1.7).

The authorization server (AS) is trusted by the protected resource to issue special-purpose security credentials—called OAuth access tokens—to clients. To acquire a token, the client first sends the resource owner to the authorization server in order to request that the resource owner authorize this client. The resource owner authenticates to the authorization server and is generally presented with a choice of whether to authorize the client making the request. The client is able to ask for a subset of functionality, or scopes, which the resource owner may be able to further diminish. Once the authorization grant has been made, the client can then request an access token from the authorization server. This access token can be used at the protected resource to access the API, as granted by the resource owner (see figure 1.8).

At no time in this process are the resource owner's credentials exposed to the client: the resource owner authenticates to the authorization server separately from any-

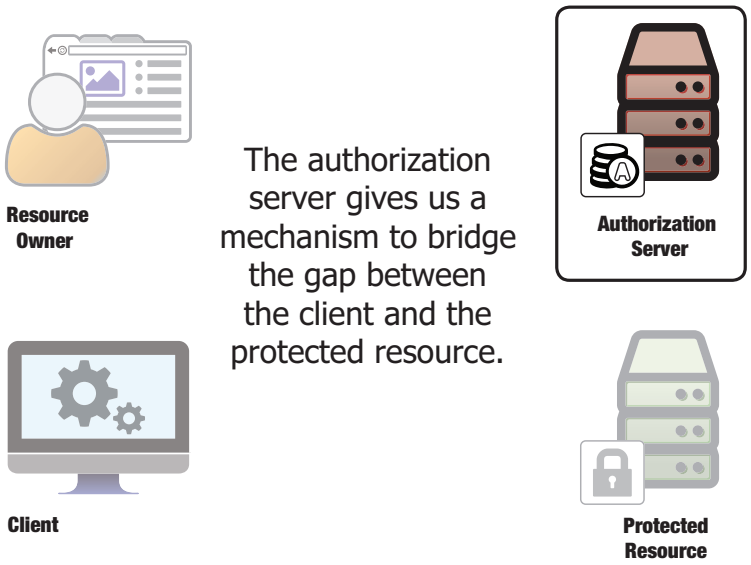


Figure 1.7 The OAuth authorization server automates the service-specific password process

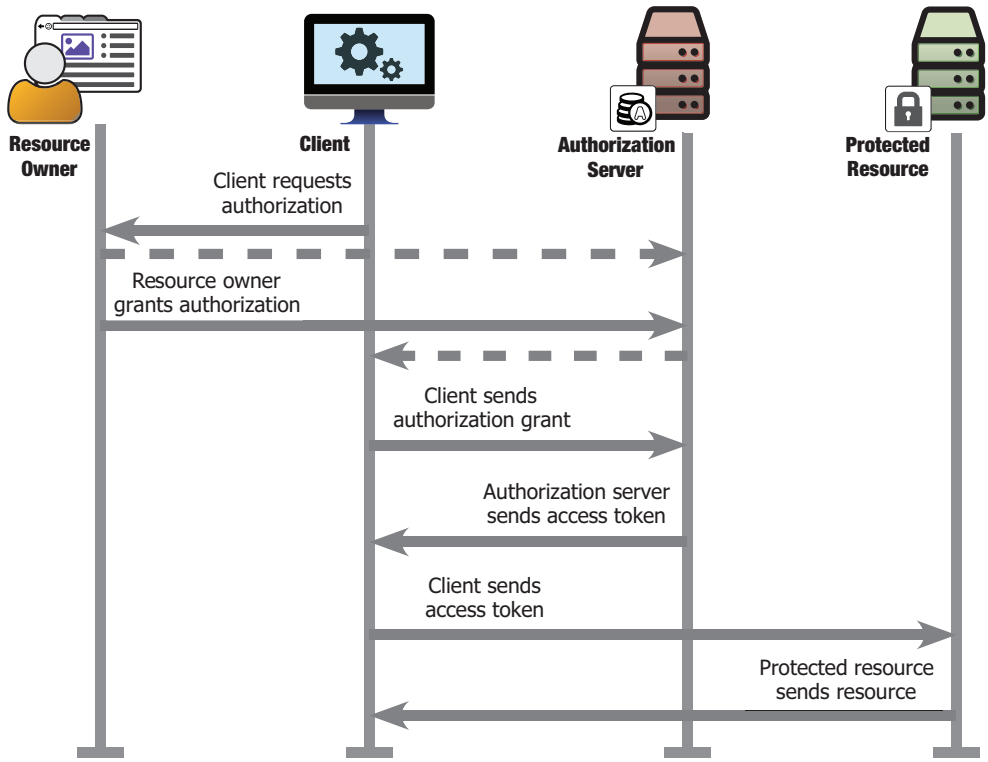


Figure 1.8 The OAuth process, at a high level

thing used to communicate with the client. Neither does the client have a high-powered developer key: the client is unable to access anything on its own and instead must be authorized by a valid resource owner before it can access any protected resources. This is true even though most OAuth clients have a means of authenticating themselves to the authorization server.

The user generally never has to see or deal with the access token directly. Instead of requiring the user to generate tokens and paste them into clients, the OAuth protocol facilitates this process and makes it relatively simple for the client to request a token and the user to authorize the client. Clients can then manage the tokens, and users can manage the client applications.

This is a general overview of how the OAuth protocol works, but in fact there are several ways to get an access token using OAuth. We'll discuss the details of this process in chapter 2 by looking in more detail at the authorization code grant type of OAuth 2.0. We'll cover other methods of getting access tokens in chapter 6.

1.3.1 Beyond HTTP Basic and the password-sharing antipattern

Many of the more "traditional" approaches listed in the previous section are examples of the password antipattern, in which a shared secret (the password) directly represents the party in question (the user). By sharing this secret password with applications, the user enables applications to access protected APIs. However, as we've shown, this is fraught with real-world problems. Passwords can be stolen or guessed, a password from one service is likely to be used verbatim on another service by the same user, and storage of passwords for future API access makes them even more susceptible to theft.

How did HTTP APIs become password-protected in the first place? The history of the HTTP protocol and its security methods is enlightening. The HTTP protocol defines a mechanism whereby a user in a browser is able to authenticate to a web page using a username and password over a protocol known as HTTP Basic Auth. There is also a slightly more secure version of this, known as HTTP Digest Auth, but for our purposes they are interchangeable as both assume the presence of a user and effectively require the presentation of a username and password to the HTTP server. Additionally, because HTTP is a stateless protocol, it's assumed that these credentials will be presented again on every single transaction.

This all makes sense in light of HTTP's origins as a document access protocol, but the web has grown significantly in both scope and breadth of use since those early days. HTTP as a protocol makes no distinction between transactions with a browser in which the user is present and transactions with another piece of software without an intermediary browser. This fundamental flexibility has been key to the unfathomable success and adoption of the HTTP protocol. But as a consequence, when HTTP started to be used for direct-access APIs in addition to user-facing services, its existing security mechanisms were quickly adopted for this new use case. This simple technological decision has contributed to the long-running misuse of continuously-presented passwords for both APIs and user-facing pages. Whereas browsers have cookies

and other session-management techniques at their disposal, the types of HTTP clients that generally access a web API do not.

OAuth was designed from the outset as a protocol for use with APIs, wherein the main interaction is outside of the browser. It usually has an end user in a browser to start the process, and indeed this is where the flexibility and power in the delegation model comes from, but the final steps of receiving the token and using it at a protected resource lie outside the view of the user. In fact, some of the key use cases of OAuth occur when the user is no longer present at the client, yet the client is still able to act on the user's behalf. Using OAuth allows us to move past the notions and assumptions of the HTTP Basic protocol in a way that's powerful, secure, and designed to work with today's API-based economy.

1.3.2 Authorization delegation: why it matters and how it's used

Fundamental to the power of OAuth is the notion of delegation. Although OAuth is often called an authorization protocol (and this is the name given to it in the RFC which defines it), it is a delegation protocol. Generally, a subset of a user's authorization is delegated, but OAuth itself doesn't carry or convey the authorizations. Instead, it provides a means by which a client can request that a user delegate some of their authority to it. The user can then approve this request, and the client can then act on it with the results of that approval.

In our printing example, the photo-printing service can ask the user, "Do you have any of your photos stored on this storage site? If so, we can totally print that." The user is then sent to the photo-storage service, which asks, "This printing service is asking to get some of your photos; do you want that to happen?" The user can then decide whether they want that to happen, deciding whether to delegate access to the printing service.

The distinction between a delegation and an authorization protocol is important here because the authorizations being carried by the OAuth token are opaque to most of the system. Only the protected resource needs to know the authorization, and as long as it's able to find out from the token and its presentation context (either by looking at the token directly or by using a service of some type to obtain this information), it can serve the API as required.

Connecting the online world

Many of the concepts in OAuth are far from novel, and even their execution owes much to previous generations of security systems. However, OAuth is a protocol designed for the world of web APIs, accessed by client software. The OAuth 2.0 framework in particular provides a set of tools for connecting such applications and APIs across a wide variety of use cases. As we'll see in later chapters, the same core concepts and protocols can be used to connect in browser applications, web services, native and mobile applications, and even (with some extension) small-scale devices in the internet of things. Throughout all of this, OAuth depends on the presence of an online and connected world and enables new things to be built on that stratum.

1.3.3 User-driven security and user choice

Since the OAuth delegation process involves the resource owner, it presents a possibility not found in many other security models: important security decisions can be driven by end user choice. Traditionally, security decisions have been the purview of centralized authorities. These authorities determine who can use a service, with which client software, and for what purpose. OAuth allows these authorities to push some of that decision-making power into the hands of the users who will ultimately be using the software.

OAuth systems often follow the principle of TOFU: Trust On First Use. In a TOFU model, the first time a security decision needs to be made at runtime, and there is no existing context or configuration under which the decision can be made, the user is prompted. This can be as simple as "Connect a new application?" although many implementations allow for greater control during this step. Whatever the user experience here, the user with appropriate authority is allowed to make a security decision. The system offers to remember this decision for later use. In other words, the first time an authorization context is met, the system can be directed to trust the user's decision for later processing: Trust On First Use.

Do I *have* to eat my TOFU?

The Trust On First Use (TOFU) method of managing security decisions is not required by OAuth implementations, but it's especially common to find these two technologies together. Why is that? The TOFU method strikes a good balance between the flexibility of asking end users to make security decisions in context and the fatigue of asking them to make these decisions constantly. Without the "Trust" portion of TOFU, users would have no say in how these delegations are made. Without the "On First Use" portion of TOFU, users would quickly become numb to an unending barrage of access requests. This kind of security system fatigue breeds workarounds that are usually more insecure than the practices that the security system is attempting to address.

This approach also presents the user's decision in terms of functionality, not security: "Do you want this client to do what it's asking to do?" This is an important distinction from more traditional security models wherein decision makers are asked ahead of time to demarcate what isn't permissible. Such security decisions are often overwhelming for the average user, and in any event the user cares more about what they're trying to accomplish instead of what they're trying to prevent.

Now this isn't to say that the TOFU method must be used for all transactions or decisions. In practice, a three-layer listing mechanism offers powerful flexibility for security architects (figure 1.9).

The whitelist determines known-good and trusted applications, and the blacklist determines known-bad applications or other negative actors. These are decisions that

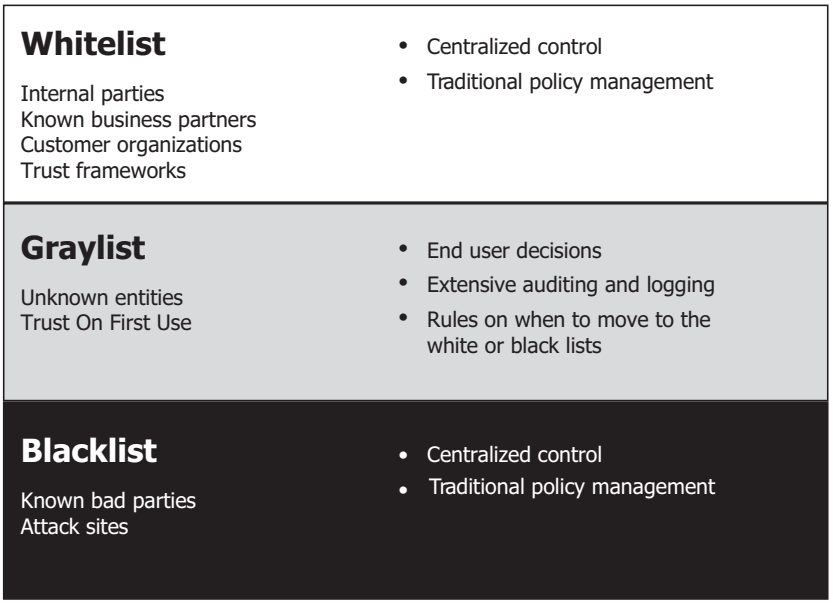


Figure 1.9 **Different levels of trust, working in parallel**

can easily be taken out of the hands of end users and decided a priori by system policy. In a traditional security model, the discussion would stop here, since everything not on the whitelist is automatically on the blacklist by default. However, with the addition of the TOFU method, we can allow a graylist in the middle of these two, an unknown area in which user-based runtime trust decisions can take precedence. These decisions can be logged and audited, and the risk of breach minimized by policies. By offering the graylist capability, a system can greatly expand the ways it can be used without sacrificing security.

1.4 OAuth 2.0: the good, the bad, and the ugly

OAuth 2.0 is very good at capturing a user delegation decision and expressing that across the network. It allows for multiple different parties to be involved in the security decision process, most notably the end user at runtime. It's a protocol made up of many different moving parts, but in many ways it's far simpler and more secure than the alternatives.

One key assumption in the design of OAuth 2.0 was that there would always be several orders of magnitude more clients in the wild than there would be authorization servers or protected resource servers (figure 1.10). This makes sense, as a single authorization server can easily protect multiple resource servers, and there are likely to be many different kinds of clients wanting to consume any given API. An authorization server can even have several different classes of clients that are trusted at different levels, but we'll cover that in more depth in chapter 12. As a consequence of this architectural decision, wherever possible, complexity is shifted away from clients and onto servers. This is good for client developers, as the client becomes the simplest

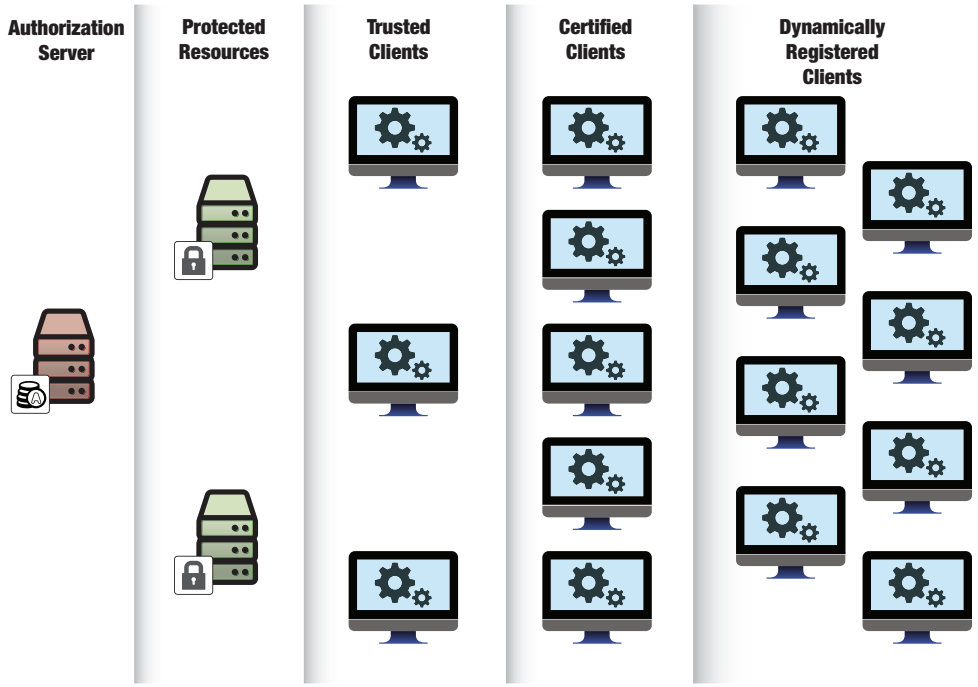


Figure 1.10 Notional relative numbers of components in an OAuth ecosystem

piece of software in the system. Client developers no longer have to deal with signature normalizations or parsing complicated security policy documents, as they would have in previous security protocols, and they no longer have to worry about handling sensitive user credentials. OAuth tokens provide a mechanism that's only slightly more complex than passwords but significantly more secure when used properly.

The flip side is that authorization servers and protected resources are now responsible for more of the complexity and security. A client needs to manage securing only its own client credentials and the user's tokens, and the breach of a single client would be bad but limited in its damage to the users of that client. Breaching the client also doesn't expose the resource owner's credentials, since the client never sees them in the first place. An authorization server, on the other hand, needs to manage and secure the credentials and tokens for all clients and all users on a system. Although this does make it more of a target for attack, it's significantly easier to make a single authorization server highly secure than it is to make a thousand clients written by independent developers just as secure.

The extensibility and modularity of OAuth 2.0 form one of its greatest assets, since it allows the protocol to be used in a wide variety of environments. However, this same flexibility leads to basic incompatibility problems between implementations. OAuth leaves many pieces optional, which can confuse developers who are trying to implement it between two systems.

Even worse, some of the available options in OAuth can be taken in the wrong context or not enforced properly, leading to insecure implementations. These kinds of vulnerabilities are discussed at length in the OAuth Threat Model Document[3] and the vulnerabilities section of this book (chapters 7, 8, 9, and 10). Suffice it to say, the fact that a system implements OAuth, and even implements it correctly according to the spec, doesn't mean that this system is secure in practice.

Ultimately, OAuth 2.0 is a good protocol, but it's far from perfect. We will see its replacement at some point in the future, as with all things in technology, but no real contender has yet emerged as of the writing of this book. It's just as likely that OAuth 2.0's replacement will end up being a profile or extension of OAuth 2.0 itself.

1.5 What OAuth 2.0 isn't

OAuth is used for many different kinds of APIs and applications, connecting the online world in ways never before possible. Even though it's approaching ubiquity, there are many things that OAuth is *not*, and it's important to understand these boundaries when understanding the protocol itself.

Since OAuth is defined as a framework, there has historically been some confusion regarding what "counts" as OAuth and what does not. For the purposes of this discussion, and truly for the purposes of this book, we're taking OAuth to mean the protocol defined by the core OAuth specification,[4] which details several ways of getting an access token. We're also including the use of bearer tokens as defined in the attendant specification,[5] which dictates how to *use* this particular style of token. These two actions—how to get a token and how to use a token—are the fundamental parts of OAuth. As we'll see in this section, there are a number of other technologies in the wider OAuth ecosystem that work together with the core of OAuth to provide greater functionality than what is available from OAuth itself. We contend that this ecosystem is evidence of a healthy protocol and shouldn't be conflated with the protocol itself.

OAuth isn't defined outside of the HTTP protocol. Since OAuth 2.0 with bearer tokens provides no message signatures, it is not meant to be used outside of HTTPS (HTTP over TLS). Sensitive secrets and information are passed over the wire, and OAuth requires a transport layer mechanism such as TLS to protect these secrets. A standard exists for presenting OAuth tokens over Simple Authentication and Security Layer (SASL)–protected protocols,[6] there are new efforts to define OAuth over Constrained Application Protocol (CoAP),[7] and future efforts could make parts of the OAuth process usable over non-TLS links (such as some discussed in chapter 15). But even in these cases, there needs to be a clear mapping from the HTTPS transactions into other protocols and systems.

---

[3] RFC 6819 https://tools.ietf.org/html/rfc6819
[4] RFC 6749 https://tools.ietf.org/html/rfc6749
[5] RFC 6750 https://tools.ietf.org/html/rfc6750
[6] RFC 7628 https://tools.ietf.org/html/rfc7628
[7] https://tools.ietf.org/html/draft-ietf-ace-oauth-authz

OAuth isn't an authentication protocol, even though it can be used to build one. As we'll cover in greater depth in chapter 13, an OAuth transaction on its own tells you nothing about who the user is, or even if they're there. Think of our photo-printing example: the photo printer doesn't need to know who the user is, only that *somebody* said it was OK to download some photos. OAuth is, in essence, an ingredient that can be used in a larger recipe to provide other capabilities. Additionally, OAuth uses authentication in several places, particularly authentication of the resource owner and client software to the authorization server. This embedded authentication does not itself make OAuth an authentication protocol.

OAuth doesn't define a mechanism for user-to-user delegation, even though it is fundamentally about delegation of a user to a piece of software. OAuth assumes that the resource owner is the one that's controlling the client. In order for the resource owner to authorize a different user, more than OAuth is needed. This kind of delegation is not an uncommon use case, and the User Managed Access protocol (discussed in chapter 14) uses OAuth to create a system capable of user-to-user delegation.

OAuth doesn't define authorization-processing mechanisms. OAuth provides a means to convey the fact that an authorization delegation has taken place, but it doesn't define the contents of that authorization. Instead, it is up to the service API definition to use OAuth's components, such as scopes and tokens, to define what actions a given token is applicable to.

OAuth doesn't define a token format. In fact, the OAuth protocol explicitly states that the content of the token is completely opaque to the client application. This is a departure from previous security protocols such as WS-*, Security Assertion Markup Language (SAML), or Kerberos, in which the client application needed to be able to parse and process the token. However, the token still needs to be understood by the authorization server that issues it and the protected resource that accepts it. Desire for interoperability at this level has led to the development of the JSON Web Token (JWT) format and the Token Introspection protocol, discussed in chapter 11. The token itself remains opaque to the client, but now other parties can understand its format.

OAuth 2.0 defines no cryptographic methods, unlike OAuth 1.0. Instead of defining a new set of cryptographic mechanisms specific to OAuth, the OAuth 2.0 protocol is built to allow the reuse of more general-purpose cryptographic mechanisms that can be used outside of OAuth. This deliberate omission has helped lead to the development of the JSON Object Signing and Encryption (JOSE) suite of specifications, which provides general-purpose cryptographic mechanisms that can be used alongside and even outside OAuth. We'll see more of the JOSE specifications in chapter 11 and apply them to a message-level cryptographic protocol using OAuth Proof of Possession (PoP) tokens in chapter 15.

OAuth 2.0 is also not a single protocol. As discussed previously, the specification is split into multiple definitions and flows, each of which has its own set of use cases. The core OAuth 2.0 specification has somewhat accurately been described as a security protocol generator, because it can be used to design the security architecture for many different use cases. As discussed in the previous section, these systems aren't necessarily compatible with each other.

Code reuse between different OAuth flows

In spite of their wide variety, the different applications of OAuth do allow for a large amount of code reuse between very different applications, and careful application of the OAuth protocol can allow for future growth and flexibility in unanticipated directions. For instance, assume that there are two back end systems that need to talk to each other securely without referencing a particular end user, perhaps doing a bulk data transfer. This could be handled in a traditional developer API key because both the client and resource are in the same trusted security domain. However, if the system uses the OAuth client credentials grant (discussed in chapter 6) instead, the system can limit the lifetime and access rights of tokens on the wire, and developers can use existing OAuth libraries and frameworks for both the client and protected resource instead of something completely custom. Since the protected resource is already set up to process requests protected by OAuth access tokens, at a future point when the protected resource wants to make its data available in a per-user delegated fashion, it can easily handle both kinds of access simultaneously. For instance, by using separate scopes for the bulk transfer and the user-specific data, the resource can easily differentiate between these calls with minimal code changes.

Instead of attempting to be a monolithic protocol that solves all aspects of a security system, OAuth focuses on one thing and leaves room for other components to play their parts where it makes more sense. Although there are many things that OAuth is not, OAuth does provide a solid basis that can be built on by other focused tools to create more comprehensive security architecture designs.

1.6 Summary

OAuth is a widely used security standard that enables secure access to protected resources in a fashion that's friendly to web APIs.

- OAuth is about *how to get a token* and *how to use a token*.
- OAuth is a delegation protocol that provides authorization across systems.
- OAuth replaces the password-sharing antipattern with a delegation protocol that's simultaneously more secure and more usable.
- OAuth is focused on solving a small set of problems and solving them well, which makes it a suitable component within larger security systems.

Ready to learn about how exactly OAuth accomplishes all of this on the wire? Read on for the details of The OAuth Dance.

In this chapter, you'll get a bird's-eye view of the Angular framework, exploring topics including generating bundles and JIT (just-in-time) vs. AOT (ahead-of-time) compilation. You'll also build a project using the time-saving Angular CLI (Command Line Interface).

Chapter 1 from *Angular Development with TypeScript, Second Edition* by Yakov Fain and Anton Moiseev

Introducing Angular

This chapter covers

- A high-level overview of the Angular framework
- Generating a new project with Angular CLI
- Getting started with Angular modules and components
- Introducing the sample application ngAuction

Angular is an open source JavaScript framework maintained by Google. It's a complete rewrite of its popular predecessor, AngularJS. The first version of Angular was released in September 2016 under the name Angular 2. Shortly after, the digit *2* was removed from the name, and now it's just *Angular*. Twice a year, the Angular team make major releases of this framework. Future releases will include new features, perform better, and generate smaller code bundles, but the architecture of the framework most likely will remain the same.

Angular applications can be developed in JavaScript (using the syntax of ECMAScript 5 or later versions) or TypeScript. In this book, we use TypeScript; we explain our reasons for this in appendix B.

NOTE In this book, we expect you to know the syntax of JavaScript and HTML and to understand what web applications consist of. We also assume that you know what CSS is. If you're not familiar with the syntax of

TypeScript and the latest versions of ECMAScript, we suggest you read appendixes A and B first, and then continue reading from this chapter on. If you're new to developing using Node.js tooling, read appendix C.

NOTE All code samples in this book are tested with Angular 6 and should work with Angular 7 without any changes. You can download the code samples from https://github.com/Farata/angulartypescript. We provide instructions on how to run each code sample starting in chapter 2.

This chapter begins with a brief overview of the Angular framework. Then we'll start coding—we'll generate our first project using the Angular CLI tool. Finally, we'll introduce the sample application ngAuction that you'll build while reading this book.

1.1 Why select Angular for web development?

Web developers use different JavaScript frameworks and libraries, and the most popular are Angular, React, and Vue.js. You can find lots of articles and blog posts comparing them, but such comparisons aren't justified, because React and Vue.js are libraries that don't offer a full solution for developing and deploying a complete web application, whereas Angular does offer that full solution.

If you pick React or Vue.js for your project, you'll also need to select other products that support routing, dependency injection, forms, bundling and deploying the app, and more. In the end, your app will consist of multiple libraries and tools picked by a senior developer or an architect. If this developer decides to leave the project, finding a replacement won't be easy because the new hire may not be familiar with all the libraries and tools used in the project.

The Angular framework is a platform that includes all you need for developing and deploying a web app, batteries included. Replacing one Angular developer with another is easy, as long as the new person knows Angular.

From a technical perspective, we like Angular because it's a feature-complete framework that you can use to do the following right out of the box:

- Generate a new single-page web app in seconds using Angular CLI
- Create a web app that consists of a set of components that can communicate with each other in a loosely coupled manner
- Arrange the client-side navigation using the powerful router
- Inject and easily replace *services*, classes where you implement data communication or other business logic
- Arrange state management via injectable singleton services
- Cleanly separate the UI and business logic
- Modularize your app so only the core functionality is loaded on app startup, and other modules are loaded on demand
- Creating modern-looking UIs using the Angular Material library

- Implement reactive programming where your app components don't pull data that may not be ready yet, but subscribe to a data source and get notifications when data is available

Having said that, we need to admit that there is one advantage that React and Vue.js have over Angular. Although Angular is a good fit for creating single-page apps, where the entire app is developed in this framework, the code written in React and Vue.js can be included into any web app, regardless of what other frameworks were used for development of any single-page or multipage web app.

This advantage will disappear when the Angular team releases a new module currently known as `@angular/elements` (see https://github.com/angular/angular/tree/master/packages/elements). Then you'll be able to package your Angular components as custom elements (see https://developer.mozilla.org/en-US/docs/Web/Web_Components/Custom_Elements) that can be embedded into any existing web app written in JavaScript, with or without any other libraries.

1.2 Why develop in TypeScript and not in JavaScript?

You may be wondering, why not develop in JavaScript? Why do we need to use another programming language if JavaScript is already a language? You wouldn't find articles about additional languages for developing Java or C# applications, would you?

The reason is that developing in JavaScript isn't overly productive. Say a function expects a `string` value as an argument, but the developer mistakenly invokes it by passing a numeric value. With JavaScript, this error can be caught only at runtime. Java or C# compilers won't even compile code that has mismatching types, but JavaScript is a dynamically typed language and the type of a variable can be changed during runtime.

Although JavaScript engines do a decent job of guessing the types of variables by their values, development tools have a limited ability to help you without knowing the types. In mid- and large-size applications, this JavaScript shortcoming lowers the productivity of software developers.

On larger projects, good IDE context-sensitive help and support for refactoring are important. Renaming all occurrences of a variable or function name in statically typed languages is done by IDEs in a split second, but this isn't the case in JavaScript, which doesn't support types. If you make a mistake in a function or a variable name, it's displayed in red. If you pass the wrong number of parameters (or wrong types) to a function, again, the errors are displayed in red. IDEs also offer great context-sensitive help. TypeScript code can be refactored by IDEs.

TypeScript follows the latest ECMAScript specifications and adds to them types, interfaces, decorators, class member variables (fields), generics, enums, the keywords `public`, `protected`, and `private`, and more. Check the TypeScript roadmap on GitHub at https://github.com/Microsoft/TypeScript/wiki/Roadmap to see what's coming in future releases of TypeScript.

TypeScript interfaces allow you to declare custom types. Interfaces help prevent compile-time errors caused by using objects of the wrong type in your application.

The generated JavaScript code is easy to read and looks like hand-written code. The Angular framework itself is written in TypeScript, and most of the code samples in the Angular documentation (see https://angular.io), articles, and blogs are use TypeScript. In 2018, a Stack Overflow developer survey (https://insights.stackoverflow.com/survey/2018) showed TypeScript as the fourth-most-loved language. If you prefer to see more scientific proof that TypeScript is more productive compared to JavaScript, read the study "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript," (Zheng Gao et al., ICSE 2017) available at http://earlbarr.com/publications/typestudy.pdf.

From the authors' real-world experience

We work for a company, Farata Systems, that over the years developed pretty complex software using the Adobe Flex (currently Apache Flex) framework. Flex is a productive framework built on top of the strongly typed, compiled ActionScript language, and the applications are deployed in the Flash Player browser plugin (a virtual machine).

When the web community started moving away from using browser plugins, we spent two years trying to find a replacement for the Flex framework. We experimented with different JavaScript-based frameworks, but the productivity of our developers seriously suffered. Finally, we saw a light at the end of the tunnel with a combination of the TypeScript language, the Angular framework, and the Angular Material UI library.

1.3 Overview of Angular

Angular is a component-based framework, and any Angular app is a tree of components (think views). Each view is represented by instances of component classes. An Angular app has one root component, which may have child components. Each child component may have its own children, and so on.

Imagine you need to rewrite the Twitter app in Angular. You could take a prototype from your web designer and start by splitting it into components, as shown in figure 1.1. The top-level component with the thick border encompasses multiple child components. In the middle, you can see a New Tweet component above two instances of the Tweet component, which in turn has child components for reply, retweet, like, and direct messaging.

A parent component can pass data to its child by binding the values to the child's component property. A child component has no knowledge of where the data came from. A child component can pass data to its parent (without knowing who the parent is) by emitting events. This architecture makes components self-contained and reusable.

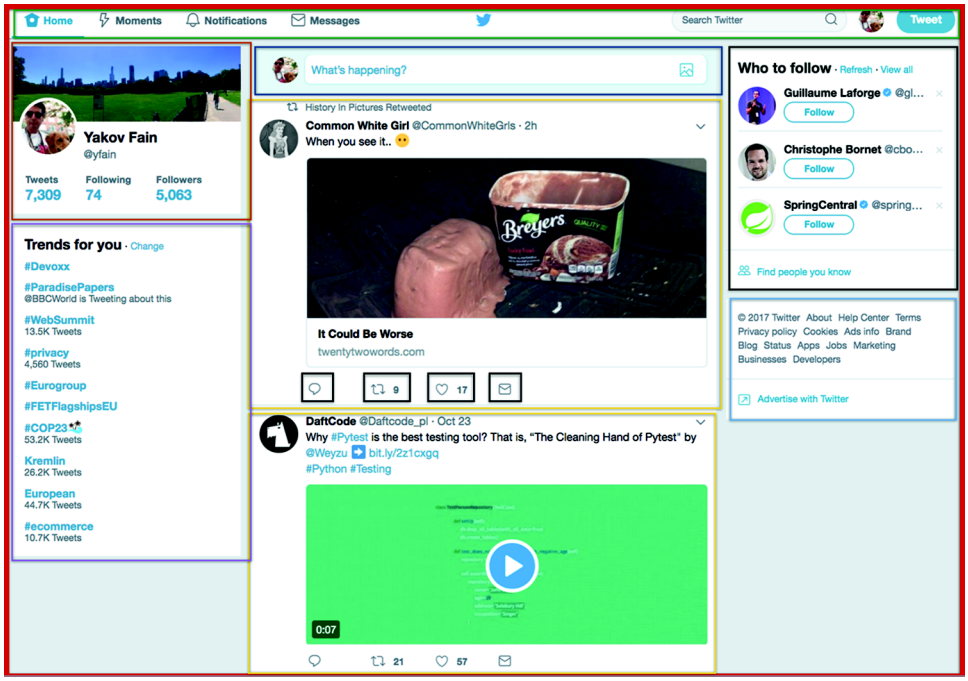


Figure 1.1 **Splitting a prototype into components**

When writing in TypeScript, a *component* is a class annotated with a decorator, `@Component()`, where you specify the component's UI (we explain decorators in section B.10, "Decorators," in appendix B).

```
@Component({
    ...
}
export class AppComponent {
    ...
})
```

Most of the business logic of your app is implemented in services, which are classes without a UI. Angular will create instances of your service classes and will inject them into your components. Your component may depend on services, and your services may depend on other services. A *service* is a class that implements some business logic. Angular injects services into your components or other services using the dependency injection (DI) mechanism we talk about in chapter 5.

Components are grouped into Angular modules. A *module* is a class decorated with `@NgModule()`. A typical Angular module is a small class that has an empty body, unless you want to write code that manually bootstraps the application—for example, if an app includes a legacy AngularJS app. The `@NgModule()` decorator lists all components

and other artifacts (services, directives, and so on) that should be included in this module. The following listing shows an example.

Listing 1.1 A module with one component

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Declares that AppComponent belongs to this module

Declares that AppComponent is a root component

To write a minimalistic Angular app, you can create one `AppComponent` and list it in the `declarations` and `bootstrap` properties of `@NgModule()`. A typical module lists several components, and the root component is specified in the `bootstrap` property of the module. Listing 1.1 also lists `BrowserModule`, which is a must for apps that run in a browser.

Components are the centerpiece of the Angular architecture. Figure 1.2 shows a high-level diagram of a sample Angular application that consists of four components and two services, all packaged inside a module. Angular injects its `HttpClient` service into your app's `Service1`, which in turn is injected into the `GrandChild1` component.

The HTML template of each component is inlined either inside the component (the `template` property of `@Component()`) or in the file referenced from the component using the `templateUrl` property. The latter option offers a clean separation between the code and the UI. The same applies to styling components. You can either

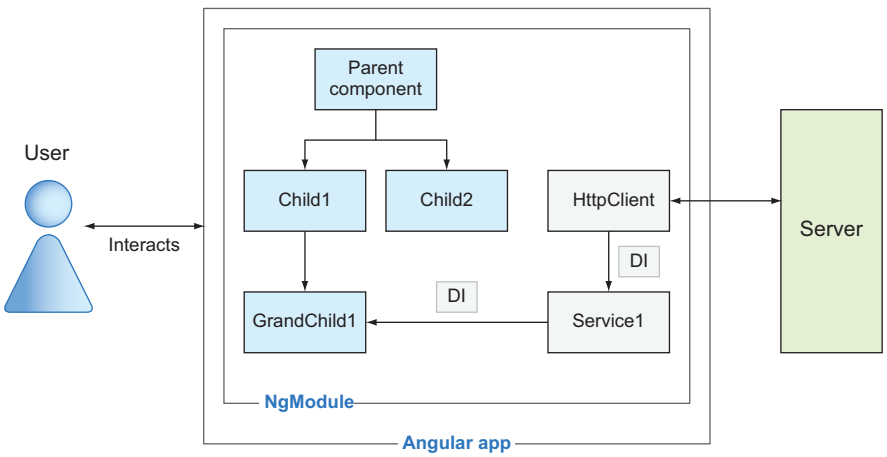


Figure 1.2 Sample architecture of an Angular app

inline the styles using the `styles` property, or provide the location of your CSS file(s) in `styleURLs`. The following listing shows the structure of some search component.

Listing 1.2 Structure of a sample component

```
@Component({
  selector: 'app-search',
   templateUrl: './search.component.html',
   styleUrls: ['./search.component.css']
 })
export class SearchComponent {
  // Component's properties and methods go here
}
```

Other components' templates can use the tag <app-search>.

The template's code is in this file.

The component's styles are in this file (there could be more than one).

The value in the `selector` property defines the name of the tag that can be used in the other component's template. For example, the root app component can include a child search component, as in the following listing.

Listing 1.3 Using the search component in the app component

```
@Component({
  selector: 'app-root',
  template: `<div>
              <app-search></app-search>
            </div>`,
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
    ...
}
```

The UI of the AppComponent includes the UI of the SearchComponent.

Listing 1.3 uses an inline template. Note the use of the backtick symbols instead of quotes for a multiline template (see section A.3 in appendix A).

The Angular framework is a great fit for developing single-page applications (SPAs), where the entire browser's page is not being refreshed and only a certain portion of the page (view) may be replacing another as the user navigates through your app. Such client-side navigation is arranged with the help of the Angular router. If you want to allocate an area within a component's UI for rendering its child components, you use a special tag, `<router-outlet>`. For example, on app start, you may display the home component in this outlet, and if the user clicks the Products link, the outlet content will be replaced by the product component.

To arrange navigation within a child component, you can allocate the `<router-outlet>` area in the child as well. Chapters 3 and 4 explain how the router works.

UI components for Angular apps

The Angular team has released a library of UI components called Angular Material (see https://material.angular.io). At the time of this writing, it has more than 30 well-designed UI components based on the Material Design guidelines (see https://material.io/guidelines). We recommend using Angular Material components in your projects, and if you need more components in addition to Angular Material, use one of the third-party libraries like PrimeNG, Kendo UI, DevExtreme, or others. You can also use the popular Bootstrap library with Angular applications, and we show how to do this in the ngAuction example in chapter 2. Starting in chapter 7, you'll rewrite ngAuction, replacing Bootstrap components with Angular Material components.

Angular for mobile devices

Angular's rendering engine is a separate module, which allows third-party vendors to create their own rendering engine that targets non-browser-based platforms. The TypeScript portion of the components remains the same, but the content of the `template` property of the `@Component` decorator may contain XML or another language for rendering native components.

For example, you can write a component's template using XML tags from the NativeScript framework, which serves as a bridge between JavaScript and native iOS and Android UI components. Another custom UI renderer allows you to use Angular with React Native, which is an alternative way of creating native (not hybrid) UIs for iOS and Android.

We stated earlier that a new Angular app can be generated in seconds. Let's see how the Angular CLI tool does it.

1.4 Introducing Angular CLI

Angular CLI is a tool for managing Angular projects throughout the entire life-cycle of an application. It serves as a code generator that greatly simplifies the process of new-project creation as well as the process of generating new components, services, and routes in an existing app. You can also use Angular CLI for building code bundles for development and production deployment. Angular CLI will not only generate a boilerplate project for you, it will also install Angular framework and all its dependencies.

Angular CLI has become a de facto way of starting new Angular projects. You'll install Angular CLI using the package manager npm. If you're not familiar with package managers, read appendix C. To install Angular CLI globally on your computer so it can be used for multiple projects, run the following command in the Terminal window:

```
npm install @angular/cli -g
```

After the installation is complete, Angular CLI is ready to generate a new Angular project.

1.4.1 Generating a new Angular project

CLI stands for *command-line interface*, and after installing Angular CLI, you can run the `ng` command from the Terminal window. Angular CLI understands many command-line options, and you can see all of them by running the `ng help` command . You'll start by generating a new Angular project with the `ng new` command. Create a new project called hello-cli:

```
ng new hello-cli
```

This command will create a directory, hello-cli, and will generate a project with one module, one component, and all required configuration files including the package.json file, which includes all project dependencies (see appendix C for details). After generating these files, Angular CLI will start npm to install all dependencies specified in package.json. When this command completes, you'll see a new directory, hello-cli, as shown in figure 1.3.

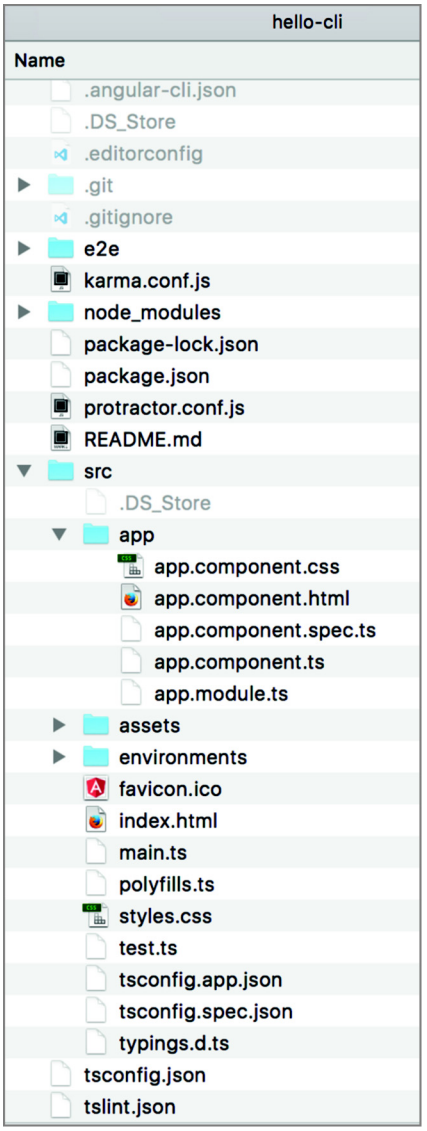


Figure 1.3 A newly generated Angular project

TIP Say you have an Angular 5 project and want to switch to the latest version of Angular. You don't need to modify dependencies in the package.json file manually. Run the `ng update` command, and all dependencies in package.json will be updated, assuming you have the latest version of Angular CLI installed. The process of updating your apps from one Angular version to another is described at https://update.angular.io.

We'll review the content of the hello-cli directory in chapter 2, but let's build and run this project. In the Terminal window, change to the hello-cli directory and run the following command:

```
ng serve
```

Angular CLI will spend about 10–15 seconds to compile TypeScript into JavaScript and build the application bundles. Then Angular CLI will start its dev server, ready to serve this app on port 4200. Your terminal output may look like figure 1.4.

```
/Users/yfain11/hello-cli
MacBook-Pro-8:hello-cli yfain11$ ng serve
** NG Live Development Server is listening on localhost:4200, open your browser
on http://localhost:4200/ **
Date: 2017-11-02T10:11:19.984Z                                              -
Hash: 6a0410d7576a15d5375e
Time: 5746ms
chunk {inline} inline.bundle.js (inline) 5.79 kB [entry] [rendered]
chunk {main} main.bundle.js (main) 20.2 kB [initial] [rendered]
chunk {polyfills} polyfills.bundle.js (polyfills) 548 kB [initial] [rendered]
chunk {styles} styles.bundle.js (styles) 33.5 kB [initial] [rendered]
chunk {vendor} vendor.bundle.js (vendor) 7.02 MB [initial] [rendered]

webpack: Compiled successfully.
```

Figure 1.4 Building the bundles with `ng serve`

Now, point your Web browser at http://localhost:4200, and you'll see the landing page of your app, as shown in figure 1.5.

Congratulations! You created, configured, built, and ran your first Angular app without writing a single line of code!

The `ng serve` command builds the bundles in memory without generating files. While working on the project, you run `ng serve` once, and then keep working on

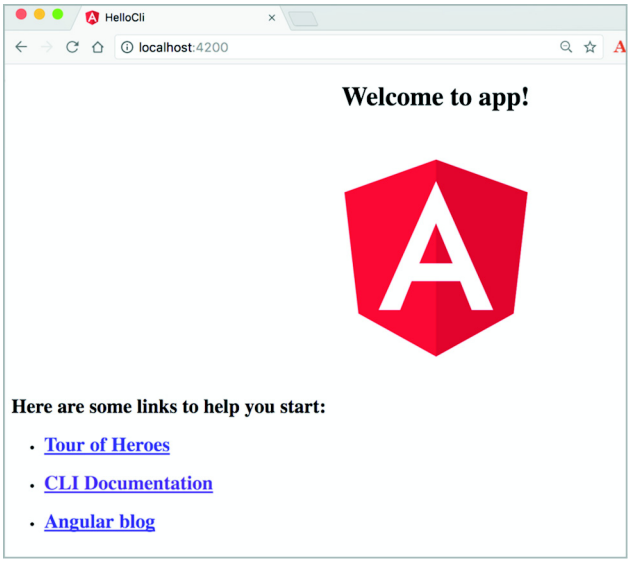


Figure 1.5 Running the app in the browser

your code. Every time you modify and save a file, Angular CLI will rebuild the bundles in memory (it takes a couple of seconds), and you'll see the results of your code modifications right away. The following JavaScript bundles were generated:

- inline.bundle.js is a file used by the Webpack loader to load other files.
- main.bundle.js includes your own code (components, services, and so on).
- polyfills.bundle.js includes polyfills needed by Angular so it can run in older browsers.
- styles.bundle.js includes CSS styles from your app.
- vendor.bundle.js includes the code of the Angular framework and its dependencies.

For each bundle, Angular CLI generates a source map file to allow debugging the original TypeScript, even though the browser will run the generated JavaScript. Don't be scared by the large size of vendor.bundle.js—it's a dev build, and the size will be substantially reduced when you build the production bundles.

Webpack and Angular CLI

Currently, Angular CLI uses Webpack (see http://webpack.js.org) to build the bundles and `webpack-dev-server` to serve the app. When you run `ng serve`, Angular CLI runs `webpack-dev-server`. Starting with Angular 7, Angular CLI offers an option to use Bazel for bundling. After the initial project build, if a developer continues working on the project, Bazel can rebuild the bundles a lot faster than Webpack.

Some useful options of ng new

When you generate a new project with the `ng new` command, you can specify an option that can change what's being generated. If you don't want to generate a separate CSS file for the application component styles, specify the inline-style option:

```
ng new hello-cli --inline-style
```

If you don't want to generate a separate HTML file for the application component template, use the inline-template option:

```
ng new hello-cli --inline-template
```

If you don't want to generate a file for unit tests, use the skip-tests option:

```
ng new hello-cli --skip-tests
```

If you're planning to implement navigation in your app, use the routing option to generate an additional module where you'll configure routes:

```
ng new hello-cli --routing
```

For the complete list of available options, run the `ng help new` command or read the Angular CLI Wiki page at https://github.com/angular/angular-cli/wiki.

1.4.2 Development and production builds

The `ng serve` command bundled the app in memory but didn't generate files and didn't optimize your Hello CLI application. You'll use the `ng build` command for file generation, but now let's start discussing bundle-size optimization and two modes of compilation.

Open the Network tab in the dev tools of your browser and you'll see that the browser had to load several megabytes of code to render this simple app. In dev mode, the size of the app is not a concern, because you run the server locally, and it takes the browser a little more than a second to load this app, as shown in figure 1.6.

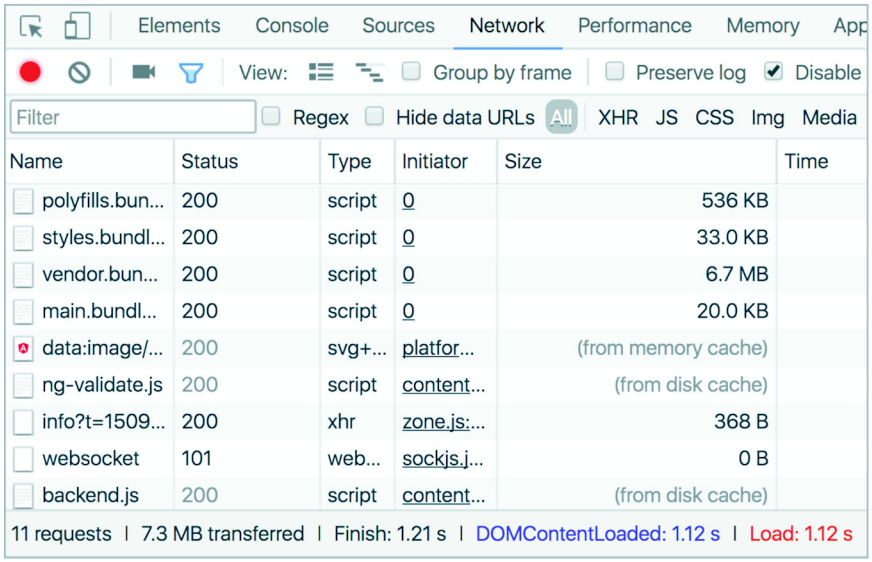


Figure 1.6 Running the non-optimized app

Now visualize a user with a mobile device browsing the internet over a regular 3G connection. It'll take 20 seconds to load the same Hello CLI app. Many people can't tolerate waiting 20 seconds for any app except Facebook (30% of the earth's population lives on Facebook). You need to reduce the size of the bundles before going live.

Applying the `--prod` option while building the bundles will produce much smaller bundles (as shown in figure 1.6) by optimizing your code. It'll rename your variables as single letters, remove comments and empty lines, and remove the majority of the unused code. Another piece of code that can be removed from app bundles is the Angular compiler. Yes, the `ng serve` command included the compiler into the vendor .bundle.js. But how are you going to remove the Angular compiler from your deployed app when you build it for production?

1.5 JIT vs. AOT compilation

Let's revisit the code of app.component.html. For the most part, it consists of standard HTML tags, but there's one line that browsers won't understand:

```
Welcome to {{title}}!
```

These double curly braces represent binding a value into a string in Angular, but this line has to be compiled by the Angular compiler (it's called `ngc`) to replace the binding with something that browsers understand. A component template can include other Angular-specific syntax (for example, structural directives `*ngIf` and `*ngFor`) that needs to be compiled before asking the browser to render the template.

When you run the `ng serve` command, the template compilation is performed inside the browser. After the browser loads your app bundles, the Angular compiler (packaged inside vendor.bundle.js) performs the compilation of the templates from main.bundle.js. This is called *just-in-time* (JIT) compilation. This term means that the compilation happens when the bundles arrive at the browser.

The drawbacks of JIT compilation include the following:

- There's an interval of time between loading bundles and rendering the UI. This time is spent on JIT compilation. For a small app like Hello CLI, this time is minimal, but in real-world apps, JIT compilation can take a couple of seconds, so the user needs to wait longer before seeing your app.
- The Angular compiler has to be included in vendor.bundle.js, which adds to the size of your app.

Using JIT compilation in production is discouraged, and you want templates to be precompiled into JavaScript before the bundles are created. This is what *ahead-of-time* (AOT) compilation is about.

The advantages of AOT compilation are as follows:

- The browser can render the UI as soon as your app is loaded. There's no need to wait for code compilation.
- The `ngc` compiler isn't included in vendor.bundle.js, and the resulting size of your app might be smaller.

Why use the word *might* and not *will*? Removing the `ngc` compiler from the bundles should always result in smaller app size, right? Not always. The compiled templates are larger than those that use a concise Angular syntax. The size of Hello CLI will definitely be smaller, as there's only one line to compile. But in larger apps with lots of views, the compiled templates may increase the size of your app so that it's even larger than the JIT-compiled app with `ngc` included in the bundle. You should use the AOT mode anyway, because the user will see the initial landing page of your app sooner.

NOTE You may be surprised by seeing `ngc` compiler errors in an app that was compiling fine with `tsc`. The reason is that AOT requires your code to be statically analyzable. For example, you can't use the keyword `private` with properties

that are used in the template, and no default exports are allowed. Fix the errors reported by the `ngc` compiler and enjoy the benefits of AOT compilation.

No matter whether you choose JIT or AOT compilation, at some point you'll decide to do an optimized production build. How do you do this?

1.5.1 Creating bundles with the –prod option

When you build bundles with the `--prod` option, Angular CLI performs code optimization and AOT compilation. See it in action by running the following command in your Hello CLI project:

```
ng serve --prod
```

Open the app in your browser and check the Network tab, as shown in figure 1.7. Now the size of the same app is only 108 KB gzipped.

Expand the column with the bundle sizes—the dev server even did the gzip compression for you. The filenames of the bundles include a hash code of each bundle. Angular CLI calculates a new hash code on each production build to prevent browsers from using the cached version if a new app version is deployed in prod.

Shouldn't you always use AOT? Ideally, you should unless you use some third-party JavaScript libraries that produce errors during AOT compilation. If you run into this problem, turn AOT compilation off by building the bundles with the following command:

```
ng serve --prod --aot false
```

Figure 1.8 shows that both the size and the load time increased compared to the AOT-compiled app in figure 1.7.

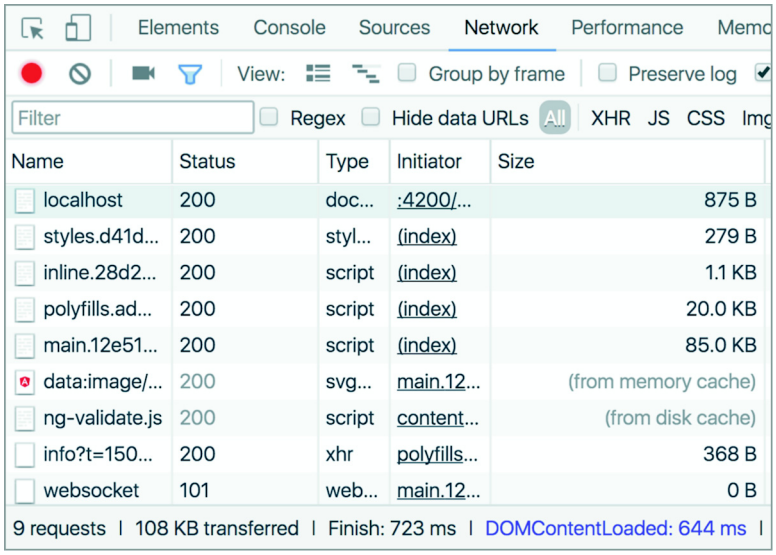


Figure 1.7 Running the optimized app with AOT

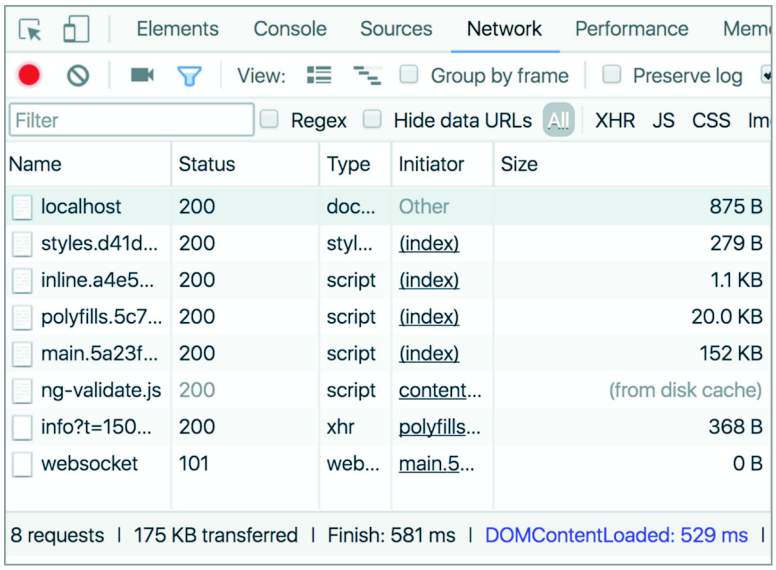


Figure 1.8 **Running the optimized app without AOT**

1.5.2 Generating bundles on the disk

You were using the `ng serve` command, which was building the bundles in memory. When you're ready to generate production files, use the `ng build` command instead. The `ng build` command generates files in the dist directory (by default), but the bundle sizes won't be optimized.

With `ng build --prod`, the generated files will be optimized but not compressed, so you'd need to apply the gzip compression to the bundles afterward. We'll go over the process of building production bundles and deploying the app on the Node.js server in section 12.5.3 of chapter 12.

After the files are built in the dist directory, you can copy them to whatever web server you use. Read the product documentation for your web server, and if you know where to deploy an index.html file in your server, this would be the place for the Angular app bundles as well.

The goal of this section was to get you started with Angular CLI, and we'll continue its coverage in chapter 2. The first generated app is rather simple and doesn't illustrate all the features of Angular; the next section will give you some ideas of how things are done in Angular.

1.6 Introducing the sample ngAuction app

To make this book more practical, we start every chapter by showing you small applications that illustrate Angular syntax or techniques, and at the end of most of the chapters you'll use the new concepts in a working application. You'll see how components and services are combined into a working application.

Imagine an online auction (let's call it ngAuction) where people can browse and search for products. When the results are displayed, the user can select a product and bid on it. The information on the latest bids will be pushed by the server to all users subscribed to such notifications.

The functionality of browsing, searching, and placing bids will be implemented by making requests to the RESTful endpoints, implemented in the server developed with Node.js. The server will use WebSockets to push notifications about the user's bid and about the bids placed by other users. Figure 1.9 depicts sample workflows for ngAuction.

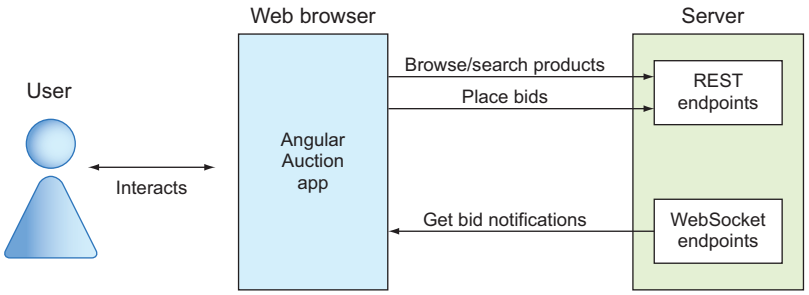


Figure 1.9 The ngAuction workflows

Figure 1.10 shows how the first version of the ngAuction home page will be rendered on desktop computers. Initially, you'll use gray placeholders instead of product images.

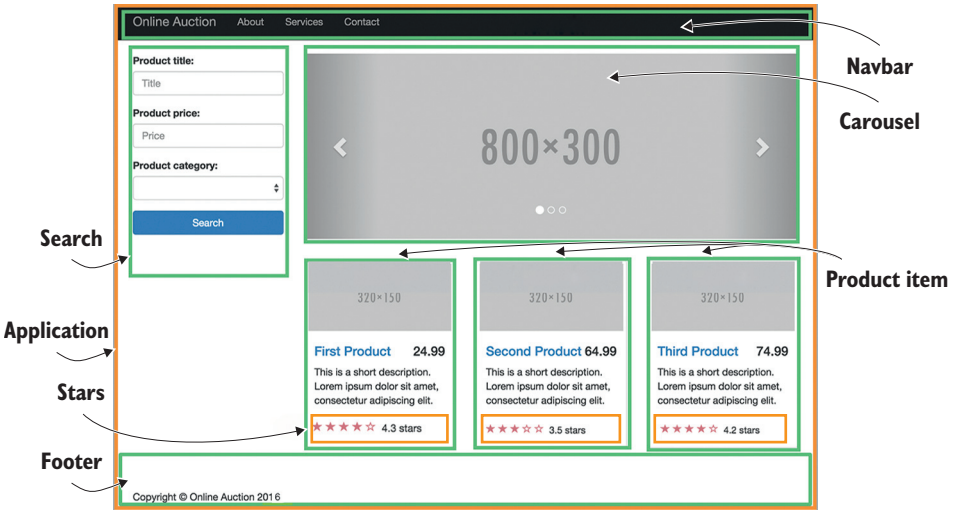


Figure 1.10 The ngAuction home page with highlighted components

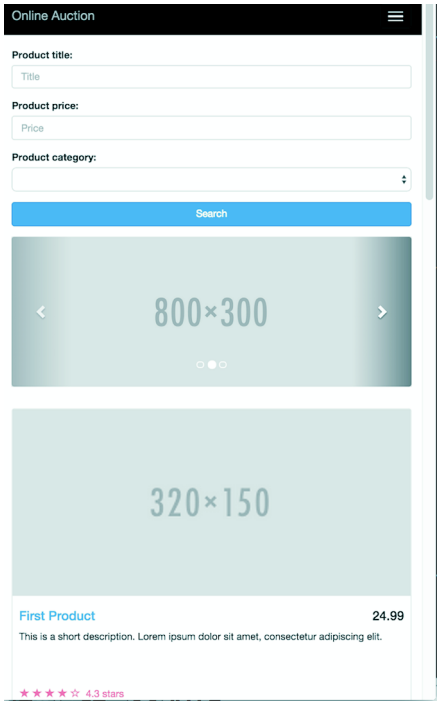


Figure 1.11 The online auction home page on smartphones

You'll use responsive UI components offered by the Bootstrap library (see http://getbootstrap.com), so on mobile devices the home page may be rendered as in figure 1.11.

Starting in chapter 7, you'll redesign ngAuction to completely remove the Bootstrap framework, replacing it with the Angular Material and Flex Layout libraries. The home page of the refactored version of ngAuction will look like figure 1.12.

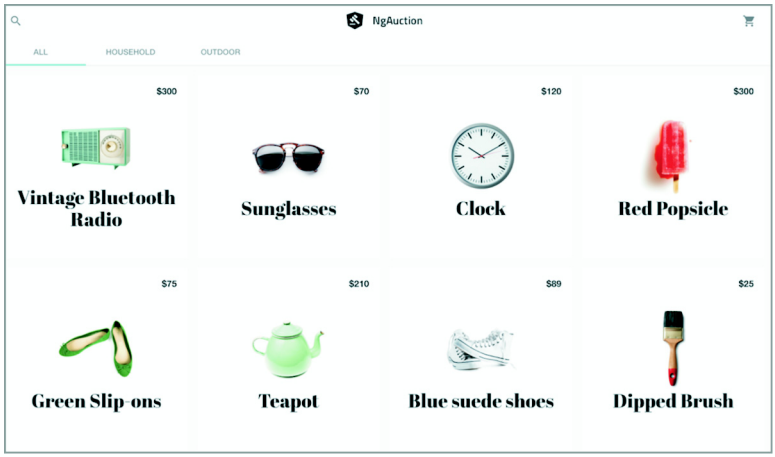


Figure 1.12 The redesigned ngAuction

The development of an Angular application comes down to creating and composing components. In chapter 2 you'll generate this project and its components and services using Angular CLI, and in chapter 7, you'll refactor its code. Figure 1.13 shows the project structure for the ngAuction app.

In chapter 2, you'll start by creating an initial version of the landing page of ngAuction, and in subsequent chapters, you'll keep adding functionality that illustrates various Angular features and techniques

NOTE We recommend that you develop Angular applications using an IDE like WebStorm (inexpensive) or Visual Studio Code (free). They offer the autocomplete feature, provide convenient search, and have integrated Terminal windows so you can do all your work inside the IDE.

Summary

- Angular applications can be developed in TypeScript or JavaScript.
- Angular is a component-based framework.
- The TypeScript source code has to be transpiled into JavaScript before deployment.
- Angular CLI is a great tool that helps in jump-starting your project. It supports bundling and serving your apps in development and preparing production builds.

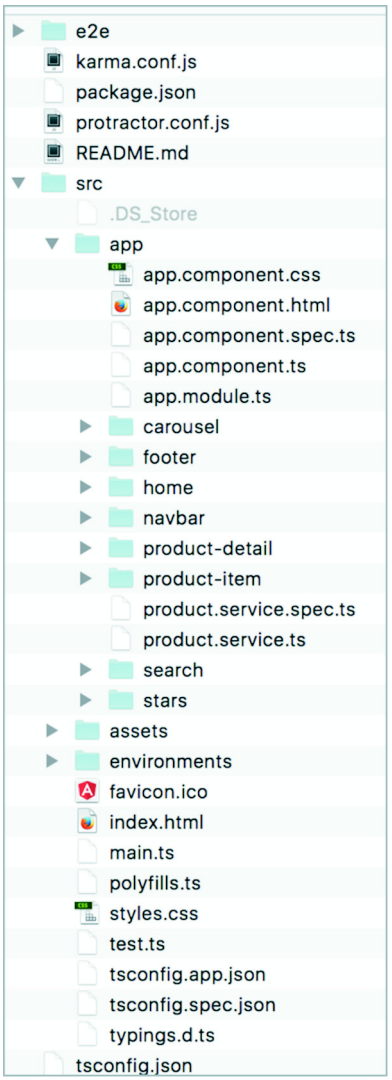


Figure 1.13 The project structure for the online auction app

index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X