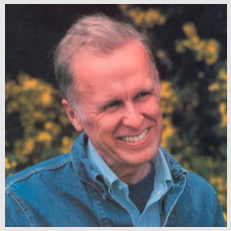
6 Client-Side Development

John Backus (1924–2007) received the 1977 Turing Award in part for "profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran," which was the first widely used high-level language.



Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.

—John Backus, quoted in IBM employee magazine *Think* in 1979

Prerequisites and Concepts

Concepts:

SaaS clients vary from small smartphones to tablets to desktop computers, representing a wide range of hardware capabilities and human interfaces. The most powerful open standards for allowing client-side user interfaces to adapt to this variation are HTML, CSS, and JavaScript. The first two were introduced in Chapter 1, and in modern browsers, can gracefully handle a surprisingly wide range of adaptations without any additional code. When HTML and CSS do not go far enough, we turn to ***JavaScript***, a dynamic, interpreted scripting language built into modern browsers. In server-centric task-focused apps, a proper understanding of what HTML, CSS, and JavaScript offer separately and together can improve an app's user experience in numerous ways.

- A browser represents a web page as a data structure called the ***Document Object Model*** (DOM), whose nodes represent both presentation elements (HTML, images, and so on) and user-input elements such as form controls.
- When a user interacts with the browser (for example, by typing, clicking, or moving the mouse) or the browser makes progress in a network interaction with a server, the browser generates an ***event*** indicating what happened. Your JavaScript code, running in the browser and associated with a specific HTML page, can take actions such as modifying the DOM (thereby changing the appearance of the page) when such events occur.
- Because JavaScript lacks concurrency facilities, event handlers must run quickly—that is, they must not block—or they will interfere with the browser's main event loop and make the UI unresponsive. Asynchronous mechanisms such as promises and continuations—collectively, asyncs in JavaScript parlance—are used to manage long-running operations without interfering with the responsiveness of the web page's UI.
- Using ***AJAX***, or Asynchronous JavaScript And XML, JavaScript code can make HTTP requests to a Web server without triggering a page reload. The information in the response can then be used to modify page elements in place, giving a richer and often more responsive user experience than traditional Web pages. Rails partials and controller actions can be readily used to handle AJAX interactions.
- While some JavaScript behaviors can be tested by Cucumber scenarios, module-level testing for more complex JavaScript is best handled by a separate TDD framework such as Jasmine[1].
- Other client-side programming scenarios include content-focused in-browser apps such as Google Docs, single-page apps (SPAs), progressive web apps (PWAs), and server-side apps. Although we do not focus on these, we discuss the pros and cons of using other tools and frameworks for working with them, including the need for more complex build pipelines.

6.1 Tooling for Client-Side Development

While this chapter will certainly cover JavaScript, equally important is the extent to which it is *not* about JavaScript. When we consider the evolution of client-side programming in the context of modern HTML and CSS as well as JavaScript, and try to understand the reasons for the proliferation of front-end frameworks such as jQuery, React, Vue, and so on, one clear lesson emerges: as with other software design problems, understanding the differences among the tools available, and the reasons they evolved, is critical in selecting the right tool for each job. A variant of Don't Repeat Yourself applies here: let's call it the DReW principle—Don't Reinvent the Wheel. Too often, developers use JavaScript to re-create features already largely provided by browsers, but with worse performance, productivity, quality, or all three.

Therefore, rather than asking whether JavaScript framework `X` is better than framework `Y`, a good developer begins by asking: Which *mechanisms* best enable my app or website to be responsive and intuitive despite varying conditions in the user's environment? A few examples of varying conditions include:

- Network speed, ranging from 10 Mbps (entry-level home broadband, mobile Internet in parts of the world) to 1 Gbps (fast broadband) in 2025
- Screen size, ranging from around 6 inches (15 cm) diagonal for smartphones to over 30 inches (75 cm) for desktop displays
- Input modality, including touchscreen, mouse, keyboard, and assistive technology such as speech-to-text

Chapter 1 introduced two of the technologies available to address this variation: ***semantic HTML*** and Cascading Stylesheets (CSS). Semantic HTML simply refers to using the HTML elements (tags) that best describe the element's logical role on the page, rather than simply its visual appearance. Doing so allows (for example) a screen reader or AI agent to navigate the page and activate specific UX behaviors for an input element, clickable button, and so on. Similarly, CSS not only controls the visual appearance of content, but also provides cues to the browser or to assistive technology that assist in site navigation and provide a more intuitive and responsive UX. Especially with modern browsers, many dynamic UI behaviors that used to require JavaScript can now be realized with HTML and CSS alone, including accordions, tooltips, and image carousels. Whenever you can achieve an effect without using JavaScript, you will reduce the total amount of code to debug and maintain, while delegating UI rednering and management to the browser, a highly tuned and optimized piece of native code, rather than to JavaScript code, whether yours or others'.

The HTML Specification[2] is constantly evolving and is the definitive list of existing elements.

That said, when JavaScript is necessary, it will often operate on the browser's ***Document Object Model*** (DOM), which this chapter also introduces. The DOM is a data structure maintained by the browser that represents the content of a web page. JavaScript, an interpreted language that runs in the browser, can both inspect and modify the DOM. Specifically:

1. JavaScript can detect a wide range of user-interaction *events* related to interactions with a web page, such as a human user clicking on a page element or typing into a text box, by associating event "listeners" with particular DOM elements.
2. By modifying the DOM—adding or removing elements, changing the content of an element (such as the text of a paragraph), or modifying the list of CSS classes associated with an element—it can change the appearance of the web page.

3. It can make HTTP requests to retrieve data from the server after a page has been loaded, possibly in response to a user-initiated event.

The first two abilities were present in the earliest versions of JavaScript. The third appeared in 1998, when Internet Explorer 5, the Microsoft browser that preceded Edge, introduced (and other browsers quickly copied) a mechanism that allowed JavaScript code to communicate with a SaaS server after a page had been loaded, without re-rendering the entire page. Importantly, because JavaScript has no concurrency facilities (as Section 6.4 explains), such requests must be asynchronous: rather than waiting for the server to respond, the programmer specifies functions that will be called when the response arrives or the request fails. The initial abstraction provided for making asynchronous calls was `XmlHttpRequest`, but it has been superseded today by the much easier to use `fetch` API and the syntactic sugar of `async` and `await` to create functions that can be suspended when a request occurs and resumed when the request completes, allowing other JavaScript code to run in the meantime.

It is easy to overlook how profound are the implications of this combination of technologies. Consider, for example, the infamous and ill-fated HTML ***<blink>*** tag. Because JavaScript code can read and modify the DOM, you could write JavaScript code that performs ***browser sniffing*** to determine whether the browser supports the tag, and if not, provide code to emulate the desired behavior. Your code would scan the DOM for elements named `blink`; start a timer of (say) 500 milliseconds; when the timer expires, make the text of all such elements invisible (or the same as the background color), and set another timer in 500 milliseconds to restore the text and start the process over again. Browsers lacking the feature could run this code, or could simply ignore the element and display non-blinking content.

UI expert Jakob Nielsen simply considered it[3] "evil."

As we describe later, such code would be an example of a ***polyfill***.

If you understand that *all front-end JavaScript frameworks essentially work this way,* you will be on your way to a big-picture view of the evolution of client-side web programming and how things have become the way they are, and correspondingly, to deciding how best to handle *your* project's client-side programming needs.

Following the "eight step plan" for learning a language originally presented in Section 2.1, Section 6.2 introduces the language's basics. Section 6.3 introduces the Document Object Model (DOM) and describes how it can be read and written by JavaScript. Section 6.4 introduces ***promises*** and asynchronous function calls ("asyncs")—a defining characteristic of JavaScript's single-threaded execution model and an implementation of the well-known ***async/await syntactic sugar***. Section 6.5 combines the preceding concepts to introduce AJAX programming, in which asynchronous calls to a SaaS server are combined with in-place page updates to enrich the client-side UI. Section 6.7 discusses testing JavaScript, which is both important and challenging since most browsers do not handle JavaScript bugs gracefully, often just silently misbehaving, or worse, freezing. You already know how to use Cucumber and Capybara for integration-level testing of JavaScript-enhanced SaaS apps, but unit-testing of JavaScript is trickier because of the runtime environment in which client-side code runs. The Jasmine TDD framework will help. Finally, the needs of some client-side applications may justify the use of a heavyweight JavaScript framework such as Vue or Angular to create single-page apps, or a non-JavaScript ("native") front-end framework such as provided by iOS or Android libraries. We discuss such frameworks in Section 6.6, and the more complex build pipelines they may entail in Section 6.8.

Summary of JavaScript background:

- The overarching goal of creating good client-side code is to provide a responsive and intuitive user experience despite widely varying conditions in the user's environment, including but not limited to network speed and screen size.
- HTML, CSS, the Document Object Model (DOM), and JavaScript, especially with the addition of asynchronous server fetches, can all be harnessed to meet this goal. While Section 6.6 describes other implementation choices that may be a better fit for some apps, as always a developer's goal is to identify the right tool for the job.
- As HTML and CSS have evolved, one use of JavaScript has been to provide support to older browsers to fully or partially emulate new and not-yet-supported features.

■ ***Elaboration: Polyfills***

As HTML, CSS, and JavaScript evolved over time, not all browsers adopted all new features at the same time. A *polyfill* is a JavaScript library intended to "fill in" an implementation of a feature not provided directly by a browser. (The name comes from *Polyfilla,* a brand of filling paste for holes in walls.) For example, the HTML5 Shiv library enabled old versions of Microsoft Internet Explorer to recognize and style new HTML5 structural elements such as `<section>` and `<nav>`, and `es5-shim` provided implementations of new ECMAScript 5 methods such as the `Array.forEach` for browsers with older JavaScript interpreters.

Self-Check 6.1.1

TBD: *need a good selfcheck question.*

◇ **TBD:** and a good answer. This can be either a concept question or a skill-building/competency question. ■

6.2 ECMAscript History and Basics

JavaScript had to "look like Java" only less so—be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JavaScript would have happened.

—Brendan Eich, creator of JavaScript

JavaScript, Microsoft JScript, and Adobe ActionScript are dialects of ***ECMAScript***, the 1997 standard that codifies the language. We follow common usage and use "JavaScript" to refer to the language generically.

Despite its name, JavaScript is unrelated to Java: LiveScript, the original name chosen by Netscape Communications Corp., was changed to JavaScript to capitalize on Java's popularity. Brendan Eich, creator of JavaScript, originally proposed embedding Scheme in the browser (Seibel 2009). Although pressure to create a Java-like syntax prevailed, many Scheme ideas survive in JavaScript, including higher-order functions, which take functions as arguments, produce a function as a return value, or both. Indeed, JavaScript is actually much more similar to Ruby. Its dynamic type system is similar to Ruby's and plays a similarly prominent role in how the language is used.

JavaScript has acquired a bad reputation that isn't entirely deserved. It began as a language intended to allow Web browsers to run simple client-side code to validate form inputs and animate page elements. Inexperienced programmers began to copy-and-paste simple

JavaScript examples to achieve appealing visual effects, albeit with terrible programming practices, giving the language itself a bad reputation. This is not to say the language has no quirks or pitfalls, but it is certainly possible to use it well.

If JavaScript has special status as the client-side language of choice, shouldn't we also write server code in it, using frameworks such as Node or Express? While it seems appealing to simply pick one language, few modern complex software systems are written entirely in a single language, because different languages solve different problems well. For example, until 2024, the Web server layer of Heroku was written in ***Erlang***, a somewhat obscure language developed for programming highly-reliable telecommunications switches,[1] because that language's abstractions are such a good match for event-driven tasks like handling multiple simultaneous connections. Using different languages for different subsystems is consistent with the design stance of microservices, in which each service optimizes one set of tasks and exposes consistent and language-agnostic APIs to other services. Managing the challenges of designing such "polyglot" (multilanguage) systems is part of the domain of software engineering.

Our fast-paced introduction to JavaScript follows the same structure proposed in Section 2.1 and is based on ECMAscript 6, which introduced new notation for classes and asynchronous functions and namespace-management mechanisms for managing multi-file projects. This section quickly introduces the elements that you probably find familiar about the language—syntax, primitive types and variable naming, control flow, and so on, as Figure 6.2 summarizes. An excellent resource to lend depth to this brief overview is the JavaScript documentation maintained by the Mozilla Developer Network[5].

Types and typing. There are only a few primitive (built-in) types: `String` (Unicode), `Number` (64-bit double precision floating point), `undefined` (having no value), `null` (a specific value different from `undefined`), `Boolean` (either `true` or `false`), and `BigInt` (rarely needed, for expressing integers of arbitrary magnitude that would overflow the `Number` type). The `Symbol` type behaves similar to Ruby's but is rarely used.

Variable Names and Scopes. As in Ruby, variables don't have types, but the objects they refer to do, so the same variable can refer to objects of different types at different times (though, as in Ruby, that's usually a bad idea). Variable names must start with a letter, underscore, or dollar sign, can also include digits, and idiomatically use `UpperCamelCase` or `lowerCamelCase` naming. A variable declaration preceded by `var` or `let` declares and optionally initializes the variable, as in `var s="Hello world"`, and sets the scope of that variable to be its enclosing block. `let` signals the JavaScript interpreter that the variable is likely to be reassigned later. Using `const` makes it an error to reassign the variable later. Using `var`, which was the only option in older versions of JavaScript, leaves it ambiguous but may prevent the interpreter from doing certain optimizations. Unlike Ruby, but like C, JavaScript allows blocks of code to be nested; a variable declared with `var` is visible to blocks nested inside the one in which it's declared, whereas a variable declared with `let` is not.

The most important compound type is `Object`, which is a collection of unordered key/value pairs. The keys are called *properties* or sometimes *slots*. JavaScript objects look and behave like Ruby hashes or Python dictionaries. Property names must be strings, although JavaScript syntax allows omitting quotes around those strings under some circumstances, and property values must not be `undefined`. Properties can be added or removed after an object is created. JavaScript allows you to express *object literals* by specifying their properties and values directly, as Figure 6.1 shows.

[1] It was recently rewritten in Go.[4]

https://gist.github.com/707970f51340686283f71858a4f02c7a

```
1  let potatoReview =
2  {
3    "potatoes": 5,
4    "reviewer": "armandofox",
5    "movie": {
6      "title": "Casablanca",
7      "release_date":  "1942-11-26T07:00:00Z"
8    }
9  };
10 potatoReview['potatoes']  // => 5
11 potatoReview['movie'].title    // => "Casablanca"
12 potatoReview.movie.title     // => "Casablanca"
13 potatoReview['movie']['title'] // => "Casablanca"
14 potatoReview['blah']          // => undefined
15 for (attr in potatoReview.movie) {
16     console.log(`${attr} is ${potatoReview.movie[attr]}`);
17 }
```

Figure 6.1: JavaScript notation for object literals, that is, objects you specify by enumerating their properties and values explicitly. If the property name is a legal JavaScript variable name, quotes can be omitted or the idiomatic dot-notation shortcut (lines 11–12) can be used, although quotes are always required around all strings when an object is expressed in JSON format; hence only lines 2–9 (minus the trailing semicolon on line 9) are legal JSON. Since objects can contain other objects, hierarchical data structures can be built (line 5) and traversed (lines 12–13). `for (`*var* `in` *obj*`) {...}` iterates over *obj*'s property names in arbitrary order (lines 15–17).

A property of the JavaScript runtime environment is the existence of a *global object* that serves as the top-level scope: Any identifier declared using `var` outside of a function becomes a property of the global object. The global object is different in different language contexts. When JavaScript runs in a browser, the global object is the `window` representing the current browser tab or view, shared by the group of related documents that have the same server origin. Among other things, `window` defines functions for DOM interaction and the browser's JSAPI (Section 6.3) as well as constants such as `NaN` and `undefined`.

Finally, JavaScript has `Array`s that can be indexed numerically, but they are actually implemented as objects (hashes) in which there is a particular relationship between property names that are integers and the array's `length` property. Like Ruby, JavaScript provides constructs to iterate over collections, abstracting away the details of how the collections are organized.

Figure 6.2 shows JavaScript's basic syntax and constructs, which should look familiar to Java and Ruby programmers. The Fallacies & Pitfalls section describes several JavaScript pitfalls associated with the figure; read them carefully after you've finished this chapter, or you may be tripped up by a JavaScript mechanism that looks and works almost but not quite like its Ruby counterpart, such as JavaScript's and Ruby's differing interpretations of "truthiness" for objects such as `null`, `undefined`, empy strings, and so on.

Functions are first-class objects, and are closures that carry their environment around with them, allowing them to execute properly at a different place and time than where they were defined, a key property for asyncs (as we will see). Just as anonymous blocks (`do...end`) are ubiquitous in Ruby, anonymous functions (`function() {...}`) are ubiquitous in JavaScript. *Arrow functions* are a shorthand way to define anonymous functions with a few limitations: they don't have their own bindings to `this`, arguments, or `super`, cannot be used as constructors, and cannot use `yield` within their body.

Classes behave much the same way as in other OO languages with the caveat that JavaScript actually does not have true classes at all. Instead, it relies on a mechanism called ***prototypal inheritance***, in which inheritance is implemented by reusing the behaviors of

Types	`typeof` *x* returns a string representation of *x*'s primitive type: one of `"object", "string", "array", "number", "boolean", "function", "undefined"`. *All numbers are doubles.*
`this`	In an object instance method, the object to which the method belongs. In an event handler (Section 6.3, the DOM element that triggered the event. In a standalone function (not an instance method of any object), refers to the *global object* (Section 6.3). In an arrow function, inherits its value from the enclosing lexical scope.

Strings	`"string", 'also a string', 'joining'+'strings'`
	`` `Catch-${11*Math.sqrt(4)}`=="Catch-22" ``
	`'mad, mad world'.slice(3,8)==", mad";`
	`'mad, mad world'.slice(-3)=="rld"`
	`'mad'.indexOf('d')==2, 'mad'.charAt(2)=='d',`
	`'mad'.charCodeAt(4)==100`

Regexps	`'mad, mad world'.split(/[, ]+/) == ["mad","mad","world"]`
	`'mad'.replace(/(\w)$/,'$1$1er')=="madder"`
	`/`*regexp*`/.exec(`*string*`)` if no match returns `null`, if match returns array whose zeroth element is whole string matched and additional elements are parenthesized capture groups.
	string`.match(/`*regexp*`/)` does the same, *unless* the `/g` regexp modifier is present. `/`*regexp*`/.test(`*string*`)` (faster) returns `true` or `false` but no capture groups.
	Alternate constructor: `new RegExp('[Hh]e(l+)o')`

Arrays	`var a = [1, {two: 2}, 'three'] ; a[1] == {two: 2}`
	Zero-based, grow dynamically; objects whose keys are numbers (see Fallacies & Pitfalls)
	`arr.sort(function (a,b) {...})` Function returns -1, 0 or 1 for `a<b,a==b,a>b`

Numbers	`+ - / %`, also `+=`, etc., `++ --`, `Math.pow(`*num,exp*`)` `Math.round(n), Math.ceil(n), Math.floor(n)` round their argument to nearest, higher, or lower integer respectively `Math.random()` returns a random number in (0,1)

Conversions	`'catch'+22=='catch22', '4'+'11'=='411'`
	`parseInt('4oneone')==4`, `parseInt('four11')==NaN`
	`parseInt('0101',10)==101`, `parseInt('0101',2)==5`,
	`parseInt('0101')==65`
	(numbers beginning with `0` are parsed in octal by default, unless radix is specified)
	`parseFloat('1.1b23')==1.1`, `parseFloat('1.1e3')==1100`

Booleans	`false`, `null`, `undefined` (different from `null`), 0, the empty string `''`, and `NaN` (not-a-number) are *falsy*; `true` and all other values are *truthy*.
Naming	`localVar`, `FunctionName`, `GLOBAL` All are conventions; JavaScript has no specific capitalization rules. `var`, `let`, `const` all scope a variable to the function in which it appears, otherwise it becomes a global (technically, a property of the global object). Variables don't have types, but the objects they refer to do.
Control flow	`while(), for(;;), if...else if...else, ?:` (ternary operator), `switch/case, try/catch/throw, return, break` Statements separated by semicolons; interpreter tries to auto-insert "missing" ones, but this is perilous (see Fallacies & Pitfalls)

Figure 6.2: Analogous to Figure 2.2, this table summarizes basic constructs of JavaScript. See the text for important pitfalls. Whereas Ruby uses `nil` as both an explicit null value and the value returned for nonexistent instance variables, JavaScript distinguishes `undefined`, which is returned for undeclared or unassigned variables, from the special value `null` and Boolean `false`. However, all three are "falsy"—they evaluate to false in a conditional.

https://gist.github.com/u/saasbook

```
1 // TBD javascript example
```

Figure 6.3: Defining a simple function using the two traditional syntaxes and the arrow-function syntax. If an arrow function takes exactly 1 argument, the parentheses around the argument list are optional. If an arrow function's body consists of a single expression, its value becomes the implicit return value; in a block body, you must use an explicit `return` statement.

https://gist.github.com/u/saasbook

```
1  class Movie {
2    constructor(title, rating) {
3      this.title = title;
4      this.rating = rating;
5    }
6    static ratingSystem = function() { return("MPAA") }; // class method
7    titleWithRating(separator) {
8      return(`${this.title} ${separator} ${this.rating}`);
9    }
10   get titleAndRating() { // may not take args!
11     return(`${this.title} (${this.rating})`);
12   }
13 }
14 var wicked = new Movie("Wicked", "PG");
15 wicked.title                 // => "Wicked"
16 wicked.titleWithRating("/") // => "Wicked/PG"
17 wicked.titleAndRating    // => "Wicked (PG)", note no parens
18 wicked.ratingSystem      // => undefined
19 Movie.ratingSystem       // => f() ...
20 Movie.ratingSystem()     // => "MPAA"
21  // TBD: show different ways to export names; and two ways to
22  // import, one of which aliases one of the names.
23
24 const Movie = class {
25   // code as above
26 }
```

Figure 6.4: Defining a class: a constructor method (line 2), static (class) method (line 6), instance method (line 7), and getter (line 10). A getter is a special kind of instance method that cannot take arguments but can be referenced just like an attribute (line 17); whether to use them over a regular method (line 16) is a matter of taste. TBD: Show export/import syntax example; discuss alternate syntax of defining a class.

NewtonScript, the bespoke language used for programming the Apple Newton "personal digital assistant," used prototypal inheritance based on the experimental language Self.

existing objects that serve as prototypes. Most JavaScript developers need not be familiar with the (interesting but quirky) details of prototypal inheritance as long as they carefully observe JavaScript syntax for class inheritance, which are syntactically similar to Ruby's, as Figure **??** shows.

We expect that most task-oriented SaaS apps' client-side code will not be very complicated, consisting of a handful of classes focusing primarily on UI code. If your project is much more complex, ES6 standardized JavaScript's mechanisms for encapsulation, which syntactically resemble Python's package management facilities. Section 6.8 discusses these facilities along with other alternatives to JavaScript, such as TypeScript, that provide better support for complex projects but incur a substantial cost in the form of a more complex build pipeline.

The JavaScript keyword `this` is analogous to Ruby `self`: when used in the body of an instance method, it refers to the object instance receiving the method call; when used in the body of a DOM event handler (Section 6.3), it refers to the DOM element on which the event was triggered.

Summary of Client-Side JavaScript and HTML:

- Like Ruby, JavaScript is interpreted and dynamically typed. The basic object type is a hash with keys that are strings and values of arbitrary type, including other hashes.
- The fundamental JavaScript data type is an object, which is an unordered collection of property names and values, similar to a hash. Since objects can nest, they can represent hierarchical data structures. JavaScript's simple object-literal notation is the inspiration for the JSON data interchange format.
- The JavaScript interpreter runs in the context of a particular runtime environment that provides a *global object* that defines many of that environment's JavaScript-relevant characteristics. In the case of the browser, the global object refers to the window displaying the page with which this interpreter instance is associated.
- The preferred unobtrusive way to associate JavaScript with an HTML page is to include in the HTML document's `head` element one or more `script` tags whose `src` attributes give the URLs of the scripts themselves, so that the JavaScript code can be kept separate from HTML markup. The Rails helper `javascript_include_tag` generates the correct URL that takes advantage of Rails' asset pipeline.

■ ***Elaboration: The global object***

In Web workers (Section 6.6) the global object named `self` defines many of the same functions and constants as `window`, but omits those relating to UI manipulation or cookies, since it is designed to support JavaScript that runs in the background. In Node.js apps, the global object named `global` defines some core Node functions such as `setTimeout`. The constant `globalThis` always refers to the current global object, so to write code that can be portably used in different contexts, use this constant to refer to properties of the global object rather than `window`, `self`, or `global`.

Self-Check 6.2.1

Is every valid JSON object parsable by JavaScript? If not, give an example of one that isn't.

◇ Yes, every valid JSON object is a valid JavaScript object. Whereas JSON requires quotes around every slot name, JavaScript sometimes does and sometimes doesn't, but it is always safe to use quotes. ■

Self-Check 6.2.2

Give an example of a JavaScript object that isn't valid JSON, despite having all its slot names in quotes.

◇ If one of the object's slots is a function, that object would not be valid JSON, since JSON slot values are limited to simple types (numbers, strings, Booleans) and collections (arrays or other JSON objects). ■

Self-Check 6.2.3

In line 16 of Figure 6.1, could we write `potatoReview.movie.attr` instead of `potatoReview.movie[attr]`?

https://gist.github.com/b23448e3c5ddd3c066ddb3153ffa11e6

```
<head>
  <script src="/public/javascripts/runWhenLoaded.js" defer></script>
  <script src="/public/javascripts/runImmediately.js" async></script>
  <script src="/vendor/javascripts/myLibrary.js" type="module"></script>
</head>
```

https://gist.github.com/b6968a45858d79e5f5edb2a76670e1a3

```
<head><title>Update Address</title></head>
<body>
  <!-- BAD: embedding scripts directly in page, esp. in body -->
  <script>
  <!-- // BAD: "hide" script body in HTML comment
       //  (modern browsers may not see script at all)
    function checkValid() {    // BAD: checkValid is global
      if !(fieldsValid(getElementById('addr'))) {
        // BAD: > and < may confuse browser's HTML parser
        alert('>>> Please fix errors & resubmit. <<<');
    }
  // BAD: "hide" end of HTML comment (1.3) in JS comment: -->
  </script>
  <!-- BAD: using HTML attributes for JS event handlers -->
  <form onsubmit="return checkValid()" id="addr" action="/update">
    <input onchange="RP.filter_adult" type="checkbox"/>
    <!-- BAD: URL using 'javascript:' -->
    <a href="javascript:back()">Go Back</a>
  </form>
</body>
```

Figure 6.5: Top: The recommended way to load JavaScript code in HTML documents, inside the `head` element. By default, scripts run as soon as they have been loaded (or parsed, if inline), unless `defer` (line 2) tells the browser not to execute the script until the document has been parsed and the DOM set up. `async` (line 3) tells the browser to execute the script as soon as it is loaded, which may be before the DOM is ready. Line 4 causes the loaded script to be treated as a module (see Section 6.8), which implies `defer`. Bottom: Executing JavaScript by mixing it into HTML pages, deprecated but sadly common in "street JavaScript."

◇ No. If we did, the expression would try to find a property literally named `attr` of `potatoReview.movie`, and failing to find one, would return `undefined`. ■

6.3 ECMAscript and the Web: The Document Object Model

We saw in earlier chapters how to create HTML and CSS on the server and send them to the browser—the basis of ***server-side rendering***. Now we consider how to modify that HTML and CSS after it has been loaded, using JavaScript. To do this the browser provides a set of *DOM APIs*—an Object Model to manipulate the Document created by that HTML, modify CSS, read back the style and layout of that document, and listen to browser *events* as the user interacts with the document. These APIs allow JavaScript to *extend* or *replace* built-in browser behaviors to add new ones, and even create documents from scratch directly from JavaScript instead of relying on the browser to do so. These DOM APIs are *imperative:* they execute an exact set of steps to change or observe the DOM. In contrast, a *declarative* API would instead focus on the desired end result rather than exactly how to achieve it. We'll return to this distinction later in the discussion of polyfills and the declarative nature of HTML and CSS.

To run JavaScript code in the browser, *each page* in your app that wants to use JavaScript functions or variables must include the necessary JavaScript code itself, because each page spawns a new copy of the JavaScript interpreter that does not share state with any other

pages. (Most browsers now also have a "tabs" metaphor, but each tab is a separate page just as if the pages had been loaded in separate windows.) The recommended and unobtrusive way to load JavaScript is by using a `script` tag referencing the file containing the code, as Figure 6.5 shows. (Section 6.8 covers more complex scenarios involving much larger JavaScript file manifests.) Rails 7 includes the gems `jsbundling-rails` and `cssbundling-rails` to automate the preparation of the loadable JavaScript and CSS assets—a process that can be quite involved, as Section 6.8 describes. For now, it suffices to place your code in one or more `.js` files in `app/assets/javascripts`, and include a call to the Rails view helper `javascript_include_tag 'application'` in your `app/views/layouts/application.html.erb` or other layout template that is part of every page served by your app. Section 6.8 describes how the behavior of this helper differs between production and non-production environments, and how build-time preparation affects client-side loading for complex JavaScript projects with many files.

Recall that when JavaScript runs in a browser, the global object is `window`. One of its important properties is `window.document`, an object that refers to the current state of the main HTML document associated with the script. (For convenience, all objects that are properties of `window` are also in the global namespace, so you can write `document` rather than `window.document` every time.) The `document` object represents the tree of nodes corresponding to the parsed HTML: the tree structure corresponds to the nesting of the HTML tags.

Most of the DOM APIs are defined on `document`. For example, `document.body` returns a reference to the `<body>` element that is present on any document. Methods such as `document.querySelectorAll(expr)` allow you to find DOM nodes by their `id`, tag (element) type, or attributes: `expr` can be any CSS selector expression including hierarchial expressions such as `#main div .summary` or attribute-based selectors such as `input[id][name$='val']`, but not including pseudo-elements such as `::before`.

JavaScript is an event based programming language, in part to integrate with the browser's notion of various kinds of events that happen in the lifetime of a web page. These events include things like:

- The HTML has finished parsing
- All subresources for the page have finished loading
- The user just clicked on the page
- The user scrolled the page
- The user submitted a form
- The page is about to unload and navigate to a new one

Each of these events is a notification that something has just happened or is about to happen. Your JavaScript code can define functions that get called (or are said to "fire") when particular events occur; these *event listeners*, which you'll also hear referred to as *observers* or *handlers*, let you run JavaScript code that takes action in response to these events. In this ***event-based programming*** paradigm, the browser's *main event loop* waits for external events to happen and notifies the appropriate *event handler* function(s) for each. Figure **??** shows a subset of the many types of events.

Element Selection (`doc` refers to `document` or `window.document`)	
`doc.getElementById('myId')`	Select element by ID (fastest method)
`doc.querySelector('#myId')`	Select first element matching CSS selector
`doc.querySelectorAll('.myClass')`	Select all elements matching CSS selector (returns NodeList)
Class Manipulation (`elt` refers to an element returned as above)	
`elt.classList.add('active')`	Add a class to an element
`elt.classList.remove('active')`	Remove a class from an element
`elt.classList.contains('active')`	Check if element has a class (returns boolean)
`elt.classList.toggle('active')`	Toggle a class on/off
DOM Manipulation	
`oldEl.replaceWith(newEl)`	Replace an element with another element
`target.parentNode.insertBefore(newEl, target)`	Insert element before target
`target.parentNode.insertBefore(newEl, target.nextSibling)`	Insert element after target (no native insertAfter)
`elt.remove()`	Remove element from DOM
State Checking	
`elt.checked`	Check if checkbox/radio is checked
`elt.selected`	Check if option is selected
`elt.disabled`	Check if element is disabled
`elt.classList.contains('active')`	Check if element has a specific class
Values and Attributes	
`old = elt.value ; elt.value = 'new'`	Get or set value of form element
`elt.getAttribute('data-id')`	Get attribute value
`elt.setAttribute('data-id', '123')`	Set attribute value
`elt.checked = true`	Set property directly
Animations (CSS Transitions)	
`elt.style.display = 'none'`	Hide element
`elt.style.display = 'block'`	Show element
`elt.classList.add('fade-out')`	Apply CSS transition class
Animations (Web Animations API)	
`elt.animate([ {opacity: 1}, {opacity: 0} ], {duration: 400})`	Fade out animation
`elt.animate([ {height: elt.offsetHeight + 'px'}, {height: 0} ], {duration: 400})`	Slide up animation

Figure 6.6: A few of the many operations on the DOM available via JavaScript.

Window/document events	`load`: entire page and all dependencies have loaded `unload`: page being unloaded `resize`: document view is resized `scroll`: user scrolls the document or an element
Form events	`submit`: form submitted `focus/blur`: an element gains or loses focus `change`: any form element changed and element loses focus `input`: an element receives user input
Low-level mouse and touch events	`click/dblclick`: single or double click on element `mousedown/mouseup, mouseover/mouseout` on an element `mousemove`: pointer is moved while over an element `keydown/keyup`: key pressed or released `touchstart/touchmove/touchend`: finger placed on screen, dragged across screen, removed from screen

Figure 6.7: A few of the many user-interaction-related event types defined by the browser. Use higher-level events such as `change` or `input` to monitor user input, rather than low-level events such as `mousedown` or `keydown`, which may not work as expected when assistive technology is used. (There are many ways to enter text without pressing keys or using the mouse to focus on the text element.)

JavaScript can add a listener by calling the `addEventListener` method on a DOM object, specifying which event to listen to and which JavaScript code to run when the event happens. For example, `document.body.addEventListener("click", (e) => console.log(e))` will print the event object to the console on a click event that reaches the body of the document. (A click event on any DOM node will *bubble* up to the body by default, so you don't even need to listen to the event on the exact DOM object that was clicked, though the `target` property of the event object tells you this information. Calling `stopPropagation` from an event handler prevents further bubbling.)

Browsers already have default responses to many events; for example, clicking the designated Submit button on an HTML form causes the browser to submit the form contents to the server. But since JavaScript listeners run before the browser's default behavior, most of these default behaviors can be suppressed by calling `preventDefault` on the event from within the listener. **TBD:** I think need to describe how events are propagated/prioritized, otherwise it's not clear why setting a listener on the body is the right thing to do in this example For example, an event listener on the body can prevent clicks on links from navigating the page like this:

https://gist.github.com/saasbook

```
1 document.body.addEventListener("click", (e) => e.preventDefault())
```

As you can see, an anonymous closure is very convenient for defining event listeners, and this pattern appears all the time in JavaScript APIs. It's possible because closures in JavaScript are first-class in JavaScript, unlike in Ruby where they are not.

Many libraries and frameworks add shorthands and convenience methods to wrap or build upon the DOM APIs by combining standard HTML and JavaScript. For example, jQuery[6] and Bootstrap[7] provide convenience wrappers and "higher level" HTML components such as cards, carousels, and drag-and-drop widgets with built-in behaviors supported by CSS and JavaScript. The much more abstract approaches of React's virtual DOM, or the extensions to declarative HTML of HTMX, can result in JavaScript code that looks nothing like calling the imperative DOM APIs, as Section 6.6 describes.

Summary of JavaScript and the DOM:
TBD: TBD

■ ***Elaboration: Define your own custom events***

Your JavaScript code can define its own event types that other JavaScript code can listen for, just like built-in event types, by using the `CustomEvent` class. This capability is commonly used to simplify incorporating higher-level components provided by JavaScript libraries as referenced above. For example, if you write a datepicker component, you might define a `dateSelected` event that fires when the user has completed all the navigation necessary to select a date, shielding the user's main app code from dealing with the individual underlying low-level UI events involved in that navigation.

Self-Check 6.3.1

In one sentence, what is the DOM for? What is the difference between the DOM and JavaScript in web pages more generally?

◇ The DOM allows JavaScript to add additional interactivity to a web page by augmenting, observing or replacing built-in HTML and CSS behaviors. ■

6.4 Idiomatic ECMAscript: Closures, Promises, Asyncs

IE was the precursor to the Microsoft Edge web browser.

Despite the name, `XMLHttpRequest` doesn't enforce any particular return type from the server and doesn't parse XML. Section 6.5 describes the name's origin.

In 1998, Microsoft added a new function to the JavaScript global object defined by Internet Explorer (IE) 5. `XmlHttpRequest` (usually shortened to XHR) allowed JavaScript code to initiate HTTP requests to a server *without* loading a new page, and use the server's response to modify the DOM of the current page, enabling richer UIs than before. The challenge of implementing XHR was that JavaScript lacks a way to express concurrency within the language, such as threads. Therefore, if a network request to the server simply waits for a response, all aspects of the browser UI handled by JavaScript would simply freeze until the response arrives, which could take seconds or longer. To solve this, XHR makes asynchronous requests: the XHR call itself returns immediately after sending the request, the browser continues the request in the background, and when the response arrives, the browser generates an event that triggers a JavaScript function you specify (the ***callback***) to handle the response. (The details are somewhat more complex, but this is the fundamental idea.) This "split call" mechanism works because JavaScript functions are closures: since the callback function is defined in the scope of the XHR call, when the callback is called it can "see" all the variables that were visible at the time the XHR request was made.

Lines 1–9 of Figure 6.8(a) show this cumbersome syntax. The `XmlHttpRequest` object has to be configured with a callback function to be called on request completion (`onload`) and the correct HTTP route (`open`), and then sent off. The important observation is that line 8 returns immediately, as soon as the request has been queued for sending—it does not wait for anything. When the server responds, the anonymous function defined in lines 3–6 will be called. Because JavaScript functions are closures, even though the call to this anonymous function is "far away" in space and time from the place where the XHR request was sent, the anonymous function can see the `apiCall` object (line 4) and extract the response body from it. This simple example omits at least two kinds of error handling: the `onload` callback should check the HTTP status of the response to make sure it is between 200 and 299 (in-

https://gist.github.com/saasbook

```
const showMoviePopup = function(targetUrl) {
  var apiCall = new XmlHttpRequest();
  apiCall.onload = function() {
    updatePage(apiCall.responseText);
    console.log("Finished fetching movie");
  }
  apiCall.open('GET', targetUrl);
  apiCall.send();
};
```

https://gist.github.com/saasbook

```
1  const showMoviePopup = function(targetUrl) {
2    var callOptions = { method: 'GET' };
3    var apiCallPromise = fetch(targetUrl, callOptions); // returns a "thenable"
4    apiCallPromise.then(function(response) {
5      // (for error handling, check `response.ok` from server)
6      updatePage(response.body);
7      console.log("Finished fetching movie");
8    });
9  };
10 const showMoviePopup = function(targetUrl) {
11   fetch(targetUrl)
12     .then((response) => {
13       updatePage(response.body);
14       console.log("Finished fetching movie");
15     });
16 };
```

Figure 6.8: (a) Top: JavaScript code to show a progress spinner and then hide it after an AJAX call completes using XHR. The example assumes that the route is `GET targetUrl`, and that `updatePage` is a function that updates the DOM using functions such as those introduced in Section 6.3. (b) Bottom: Lines 1–9 show the use of `fetch` to achieve the same result. Lines 10–16 are a more idiomatic version of the same code. The `callOptions` argument (lines 2–3) can specify various properties of the fetch, such as the HTTP method, body payload for `PUT` or `POST` requests, and as we discuss later, a timeout handler.

https://gist.github.com/u/saasbook

```
1  // Using async/await syntactic sugar
2  const showMoviePopup = async function(targetUrl) {
3    showSpinnerAnimation();
4    var response = await fetch(targetUrl);
5    hideSpinnerAnimation();
6    if (!response.ok) {
7      throw new Error("Server response error");
8    }
9    updatePage(response.body);
10 };
11 // What a call might look like:
12 showMoviePopup(someUrl)
13   .then(() => console.log("Fetch succeeded"))
14   .catch((errorMsg) => console.error(errorMsg));
```

Figure 6.9: `async` forces a function to immediately return a promise rather than blocking. Inside an `async` function (and only there), `await` can be used to pause function execution while a long blocking operation happens.

dicating success), and take appropriate error actions otherwise; and we would normally also assign a function to `apiCall.timeout` to be called if the request times out before receiving a server response.

The clumsiness of the syntax makes it difficult to follow what should be a conceptually simple control flow: client calls server, server responds, client uses response to update the DOM. Lines 1–5 of Figure 6.8(b) show how the new `fetch` API, introduced in 2015 and now supported by all major browsers, simplifies the code by using a ***promise***—an abstraction for the result of a computation that has not yet completed. The built-in class `Promise` provides this abstraction in JavaScript, and `fetch` returns an instance of this class for which the deferred computation result is the server's response. This promise object can be passed around like any other object, but its most salient property is that you can call `then` on it, which takes two arguments: the first is a callback to run when the promise is *resolved* or *fulfilled* (the result of the computation becomes available), and the second optional one is a callback to run if the promise is *rejected* because an error prevented the computation from completing. (We discuss below what happens if a promise is rejected and no rejection callback is provided.)

Sometimes also called a *future*, *delay*, or *deferred*, promises originated in functional programming.

In the spirit of duck typing, a promise may be said to be a kind of *then-able*—an object that responds to a `then` method.

In line 2 of Figure 6.8(a), the value of `apiCall` is an instance of `XmlHttpRequest`, which expects to be passed various functions to call in the event of success, failure, timeout, and so on, and internalizes the machinery of how and when those functions get called. In contrast, in line 2 of Figure 6.8(b), the value of `apiCallPromise` is an instance of a promise, and the resolve callback is a function that accepts the server response (for simplicity, we omit error checking, as in the first `XmlHttpRequest` example) and uses it to modify the DOM.

Lines 1–7 of Figure 6.8(b) represent an improvement in conciseness over (a); lines 8–12 improve further by omitting the explicit intermediate assignment of a variable to the promise object and by using the arrow syntax for function definition. But in both cases, the conceptually simple order of operations—make network call, use response to update page, log to console—is still somewhat obfuscated. The syntactic sugar provided by `async` and `await` addresses this last limitation, as Figure 6.9 shows, by effectively "wrapping" a function call in a promise. This syntax separates the runtime control flow from the program text, which can now read like linear synchronous code.

Since `showMoviePopup` now returns a promise, its caller must be prepared to accept a `Promise` as a response (lines 13–14), just as the call to `fetch` returned a `Promise` (lines

https://gist.github.com/u/saasbook

```
// Promise chain example: chaining multiple async operations
return fetch(targetUrl)
  .then((response) => updatePage(response.body))
  .then(() => showAnimation())
  .then(() => console.log("All done"));

// Using a timeout to reject a promise:  Fetch a URL
// and either return its contents as JSON or time it out.
fetch(url, { signal: AbortSignal.timeout(5000) })
  .then(response => {
    if (!response.ok) {
      throw new Error("Server error");
    }
    return response.json();
  })
  .then(data => console.log(data))
  // If timer runs out first, the AbortSignal will throw
  // an exception that we can catch here
  .catch(error => {
    if (error.name === 'AbortError') {
      console.error("Fetch timed out");
    } else {
      console.error("Fetch error:", error);
    }
  });
```

Figure 6.10: `AbortSignal` helps convert a timeout into a promise rejection. We pass this timeout signal object to `fetch` as a call option (like line 11 of Figure 6.8).

12–14 of Figure 6.8). If `showMoviePopup` happens to be able to run to completion without blocking, the promise just resolves immediately; otherwise, the promise is either resolved if the `await fetch` completes successfully, or rejected by throwing an error if unsuccessfully.

What happens if a promise is *never* resolved or rejected? For example, perhaps the server targeted by `fetch` is unresponsive, or the client is disconnected from the Internet. Promises (and therefore `fetch`) do not by default provide any timeout behavior—the promise will remain unresolved forever, and the resources consumed by keeping the request open will continue to be held. Therefore, handling this case cleanly adds some complexity, as Figure 6.10 shows, by turning the timeout into a promise rejection.

Service workers, which we describe in Section 6.10, provide a graceful way to deal with disconnected operation.

Finally, what happens if there is no handler specified to reject a promise? If the promise is part of a promise chain **TBD:** how much do we need to go into detail about this?, the rejection is passed up the chain (similar to how most languages handle exceptions) until someone handles it or it reaches the top level. What happens at the top level depends on the runtime JavaScript environment. If it is a Web browser, typically the browser will log the error to the console log. If it is a server-side app based on Node or a Node-related framework, the app may raise an app-level exception.

- JavaScript lacks in-language concurrency, so to prevent the browser UI from freezing, long-running operations that would otherwise block progress must be "split" asynchronously.
- Initially, `XmlHttpRequest` (XHR) accomplished this splitting by returning immediately when making a call to the server, but arranging to call a programmer-specified callback function when the response arrives.
- Later, `fetch` simplified the syntax by making use of the new and more general `Promise` abstraction, which represents the result of a computation that will complete in the future, with the syntactic sugar `fetch(url).then(callback)`. `fetch` immediately returns a promise that, when resolved or rejected, will call the callback function, just as XHR would arrange to call the success or failure function when the server response arrived.
- Another new way to express "suspend this long-running operation and resume later" is to declare a function `async`, and in the body, place `await` before the long-running statement. The execution state at that point is paused, to be resumed when the promise that is the argument of `await` is resolved or rejected.
- Although both `fetch().then()` and `async/await` provide the convenience of sequential-looking code, under the hood they both use the same fundamental mechanisms as XHR—callbacks and closures—but provide those semantics more generally rather than only in the context of making network calls.

■ ***Elaboration: Asyncs are continuations***

A ***continuation*** is an abstract representation of the state of a process—you can think of it as all of the information needed to suspend the process now and resume it later, including the closure of all in-scope variables, the state of the function call stack, and so on. Declaring a function `async` tells JavaScript to call it in such a way that a continuation can be created when `await` is called, at which point JavaScript returns to the event loop, waiting for another event to happen. When the `await` computation completes (i.e. the promise is fulfilled), the continuation is used to resume the task. Hence `await` can only be used when JavaScript has been advised that this special context is set up—in the body of an "async-aware" function—and by transitivity, that function also has to have been called from async-aware environment, and so on up to the top level, or in the case of modern browsers, an event handler attached to a DOM element. Continuations also often support exception handling, user-level threads such as ***green threads***, and coroutines such as those implied by Ruby's `yield`. Some languages feature *first-class continuations*: a continuation object can be explicitly created, manipulated, passed around, jumped to, and so on. Not JavaScript, though: similar to how Rails lacks a generalized aspect-oriented programming facility but instead exposes a few specific abstractions based on AOP (Section 5.1), JavaScript lacks first-class continuations but exposes the `async/await` abstraction that relies on them.

Self-Check 6.4.1

Why can `await` only be used in the body of a function declared `async`?

https://gist.github.com/u/saasbook

```
1 async function getText(url) {
2   let response = await fetch (url);
3   if (!response.ok) {
4     throw new Error(`Response status: ${response.status}`);
5   }
6   let text = await response.text();
7   return text;
8 }
```

Figure 6.11: Calling the `fetch` API in an `async` function using `await` both to avoid blocking on the `fetch` call (line 2) and to avoid blocking while the full body of the response is downloading (line 6).

◇ `await` indicates the desire to immediately return a promise rather than stall the interpreter while performing a long-running or blocking operation. Therefore the caller of the function containing `await` must expect to be able to be handed a promise. `async` ensures that a function is defined to return a promise.

■

Self-Check 6.4.2

What do you think eventually happens if your code starts a long-running operation that is not protected by a timeout (such as in Figure 6.10) and just causes the JavaScript interpreter to remain blocked?

◇ Eventually the browser itself will detect that the interpreter is hung, and will display a message to the user such as "This page isn't responding. Do you want to keep waiting?" This is one of the few cases in which JavaScript errors are manifested overtly to the user.

■

6.5 AJAX

Up to this point, our web app examples have all been *full-page navigations*: the user types a URL or clicks on a link, an HTTP request goes to the server, and the response is a full HTML web page, which may reference CSS, images, and JavaScript subresources. Modern web apps can also use JavaScript to send *async* requests to the server for additional data and then use the response to modify the DOM, without requiring a full-page reload. We're already familiar with the essential ingredients: the ability to call functions asynchronously without blocking (Section 6.4) and the APIs for inspecting and manipulating the DOM (Section 6.3). This section assembles the ingredients to show the full workflow of requests that for historical reasons were originally called *AJAX requests*, but which we'll refer to by their more contemporary name *async fetches* from now on.

As Figure 6.11 shows, `fetch()` returns a `Promise` that, when it resolves, returns an instance of `Response`, which represents the server's response. Of particular note, while the `ok` *attribute* of the response is available immediately (line 3), getting the content of the response requires calling the (async) *function* `text()`, because at this point, the response headers have been received but not necessarily the entire response body. So `text()` returns a `Promise` that, when resolved, returns a JavaScript string representing the response body. Returning the (immediately-available) `Response` earlier also allows JavaScript to know earlier rather than later whether the request succeeded, and what its headers and metadata are.

AJAX (***Asynchronous JavaScript And XML***) originally combined async requests with server responses in XML, an emerging interchange format in the early 2000s. When HTML5 rather than ***XHTML*** became the successor to HTML, JSON superseded XML.

What kinds of data might an async fetch expect in the response? In principle, an HTTP response body can contain just about any type of data, but two types are particularly common: a JSON object and an HTML fragment. We decribe each of these cases and show how to do a simple RottenPotatoes enhancement—a "popover" showing movie information—with each method.

Having the server return a JSON object would seem to be particularly convenient for JavaScript, because JSON is essentially a serialized JavaScript object. The JavaScript function `JSON.parse()` turns a well-formed JSON string into a JavaScript data object, and `JSON.stringify()` turns a JavaScript object into a JSON string. Indeed, since JSON is so common in web apps, the `fetch` API includes an *async* `json()` method on the `Response` object that parses the response text into a JavaScript object and returns it. This method is the typical way that web apps fetch and then process pure data available on the server.

`parse` is both faster and safer than `eval`, previously the only way to convert JSON to a JavaScript object.

But once you have JSON data coming back from the server, what is the client supposed to do with it? If the data is fetched for the purpose of showing something to the user and letting them interact with it, the JavaScript code must perform *client-side rendering*— using the data to make updates to the document via the DOM APIs, essentially "converting" JSON data to HTML and injecting it into the DOM, either by replacing the HTML content of entire DOM elements or hard-coding special edits to modify DOM content in place. We note, though, that if the client-side views are normally generated from HTML templates, as is the case with Rails and many other server-side Web frameworks, the client is essentially duplicating the view-rendering logic on the server.

Don't confuse this usage of "rendering," meaning "update the DOM," with the very different meanings of "browser converts DOM to pixels" or "Rails instantiates a view template." Sadly, all three uses are standard.

Rather than this non-DRYness, why not just have the server return an HTML fragment, and let JavaScript use the DOM API calls to insert or replace that element entirely in the DOM? Rails supports exactly this approach by returning HTML partials, which you already know from Section 5.1.

What do these two approaches look like from the server's point of view? Importantly, an HTTP request resulting from `fetch` looks like any other HTTP request. For example, if the URL passed in Figure 6.11 is the home page of Rotten Potatoes, then the server response will be a string representation of the HTML for that page. But async JavaScript requests rarely expect an entire well-formed HTML page as the response; as we described above, they usually expect either a JSON object or a partial chunk of HTML. Although it is possible to set the value of the `Accept:` header in the `fetch()` API to indicate (for example) that JSON is desired, and cross-check the result by examining the `Content-type:` header of the response object, a best practice is to use *different routes* for full-page loads than for async fetch responses.

Figures 6.12 and 6.14 readily demonstrate the differences among these approaches by using each to re-implement the popover example from earlier in this section. Rotten Potatoes could expose a `GET` route for requesting the text of a specific review for some movie.

TBD: In the code examples below: would be nice to have some context showing how the JS was loaded, how the event handler was attached to a DOM element, etc. How much of the "enclosing" HTML should we show?? Also, I'd like to at least in passing illustrate that when serializing a model object, you can either have it include the ID's (only) of the associated objects, or have Rails recursively serialize the associated object(s) in place. Not sure what is the right balance for these examples.

TBD: summary here

https://gist.github.com/saasbook

```ruby
# controller action for `GET /reviews/:review_id`
# assuming @review_id has been set, and Review belongs to Moviegoer
def review_for
  @review = Review.find(@review_id)
  render :json => @review.to_json(
           # :include => {:moviegoer => {:only => [:id,:name]}},
           :except => [:updated_at])
end
```

https://gist.github.com/saasbook

```javascript
// possible JSON response, assuming a Review with simple attributes
{
  "id": 324,
  "potatoes": 5,
  "moviegoer_id": 22,
  "description": "One of the classics...",
  "created_at": "2020-08-01T12:30:05Z"
},
{
  "id": 170,
  "potatoes": 4,
  "moviegoer_id": 8,
  "description": "Je n'avais jamais vu...",
  "created_at": "2022-01-21T02:25:11Z"
}
```

https://gist.github.com/saasbook

```javascript
// Event handler attached to a "Review details" button whose HTML
// element ID is "review_NNN" where NNN is the review ID

// tbd: would be nice to have some context showing how this JS was
// loaded, how the event handler was attached to a DOM element, etc.
// how much of the "enclosing" HTML should we show??
```

Figure 6.12: Adding a "review popover" to RottenPotatoes by returning a JSON object with the review details and using client-side rendering to display it. Top: Rails' built-in `to_json` can convert most Rails objects to a sensible JSON representation. `render json:` causes the HTTP response headers to indicate that the content type is JSON. Middle: the body of a successful response is a string that can be readily converted to a JSON object. Bottom: Client-side rendering of the JSON data.

https://gist.github.com/u/saasbook

```javascript
// JSON response if line 6 of controller action is uncommented
{
  "id": 324,
  "potatoes": 5,
  "moviegoer": {
    "id": 22,
    "name": "Ben Bitdiddle"
  },
  "description": "One of the classics...",
  "created_at": "2020-08-01T12:30:05Z"
},
```

Figure 6.13: What the JSON response would look like if line 6 of the controller code in Figure 6.12(top) were uncommented.

https://gist.github.com/saasbook

```
# controller action for `GET /reviews/:review_id`
# assuming @review_id has been set, and Review belongs to Moviegoer
class ReviewsController < ApplicationController
  def review_for
    @review = Review.find(@review_id)
    render :partial => 'show_popover'
  end
end
```

https://gist.github.com/saasbook

```
<!-- intended to be the inner content of a div -->
<!-- tbd simple HTML markup for the popover -->
```

https://gist.github.com/saasbook

```
// Event handler attached to a "Review details" button whose HTML
// element ID is "review_NNN" where NNN is the review ID
// code will insert the HTML into a <div> with a predetermined ID?
```

Figure 6.14: The same popover functionality as Figure 6.12, but avoiding client-side rendering. Top: `render :partial` instantiates an existing Rails partial and sets the response's `Content-type` to HTML. Middle: the partial in `application/views/review/show_popover.html.erb`. Bottom: the client-side code simply replaces the HTML of an existing element with the supplied fragment.

■ ***Elaboration: HTMX***

Given the turbulent evolution not only of JavaScript but of the architectural approach to client-side programming, and the extent to which it was enabled by polyfills (see the Elaboration in Section 6.1, it seems perilous to single out any specific polyfill library for attention, but the extremely compact HTMX[8] library is worth mentioning (in the opinion of your authors) for its simplicity and its adherence to the Web's original declarative design. In essence, HTMX uses polyfills to allow any HTML element to submit an asynchronous HTTP request, rather than just `<a>` and `<form>`, and the request may be triggered by any event type, rather than just `click` and `submit`. The server returns a well-formed block of HTML which gets integrated directly into the page. (You can readily see how Rails' routing subsystem and use of partials make it easy for Rails apps to support HTMX.) HTMX is a polyfill that interprets attributes such as `hx-post` and `hx-target` attached to arbitrary HTML elements. In other words, it polyfills a *declarative* feature with *imperative* code: the developer using HTMX writes declarative code, even if HTMX's implementation is imperative. This allows developers to continue to use HTML in the declarative way it was intended, yet cleanly support UX patterns that rely on async fetches without a lot of boilerplate JavaScript. Of course, it remains to be seen whether HTML5 will officially incorporate such functionality, thereby obviating the need for the HTMX library.

HTMX is the successor to Intercooler.js.

Self-Check 6.5.1

TBD: *selfcheck question or (better) competency*

◇ **TBD:** answer ■

6.6 Client-Side App Architecture and Implementation

As with any software engineering project, the first step in creating a mobile Web app is knowing how to identify and then use the right tool for the job. To that end, this section

discusses tradeoffs for choice of client-side architecture and JavaScript frameworks in light of these different constraints and needs.

We can broadly classify web apps into two categories. *Task-oriented* web apps are focused on finding answers and getting a task done: examples include news/blogs, e-commerce of all kinds (anything that can be bought or sold online), encyclopedias, and question-answering. Such apps typically feature short-lived user sessions. In contrast, *content-oriented* apps are focused on creating content or providing an immersive experience in consuming it: examples include social media, video streaming, email or document authoring, and design tools. As Chapter 1 explained, as the Web became widespread, even desktop apps began to move there. It quickly became apparent that the limitations of early Web technologies would make it difficult for those apps to match the smooth user experience of their desktop counterparts. The evolution of Web technologies since then has been driven in part by the desire to close this gap. For example, desktop apps generally work well when disconnected from the Internet; they feature a user experience that updates the content of the app's windows without the "page redraw" behavior typical of early Web apps; and they feature smooth transitions between UI states, such as fades, dynamic menus, and so on. We will use the whimsical term *appiness* to refer to a high-quality experience when using such apps.

How should we architect task-oriented and content-oriented apps? Considering the task-oriented category first, server-centric apps that serve HTML and CSS to the client, possibly augmented with some JavaScript, have been the focus of the book so far and are generally more than adequate for this purpose. In task-oriented apps, client-side interactions are typically limited to standard web widgets such as forms, buttons, tooltips, and dialogs. Just a few years ago, a case could be made that making the interactions with those elements feel smooth and professional required using either a heavyweight JavaScript framework such as React or a client-specific framework such as React Native to create a ***single-page application*** (SPA). Rather than loading and rendering HTML, such apps relied on JavaScript or native code to load content from the server dynamically, and render it into HTML `<div>` containers or a client-native screen buffer, without the user ever experiencing a page reload. The goal was to give task-oriented apps a more polished UX, with animated page transitions, toggles, modal dialogs, and even carousels. But modern facilities have largely eliminated the need to use JavaScript for such features. Declarative CSS Transitions[9] animates changes to individual DOM elements on a page. The View Transition API[10] animates transitions between entire DOM states, including between pages or views. The HTML `popover` attribute on textual elements eliminates the need to use JavaScript to create a temporary, floating UI element. These and other improvements make it possible to achieve smooth effects today using plain HTML and CSS with a small amount of JavaScript for tasks such as client-side form validation and analytics. The advantage of this approach is that it allows apps to load quickly and take maximal advantage of the capabilities of highly-engineered browsers. We refer to this architecture as ***server-side rendering*** since the HTML and CSS are created on the server.

In contrast, "appiness" for content-oriented web apps has proven harder to achieve. These apps are immersive, long-lived, have bespoke UX needs, and often must work gracefully when disconnected. For these, it may make more sense to adopt a different architecture in which the client receives JSON content via a server API, stores and renders the content locally (client-side rendering), and locally converts UX events to server requests rather than involving the server in UX patterns. We refer to such an architectures as *REST-less*, since it departs significantly from the pattern of "CRUD operations that manipulate representations of resources."

https://gist.github.com/u/saasbook

```
1 // TBD: JSX/React implementation of popover
```

Figure 6.15: The React implementation of the popover example shows that even achieving something as simple as a popover involves somewhat tricky use of React *hooks* such as `useEffect` and `useState`. React also introduces the need for *transpiling* JSX to JavaScript, resulting in the need for a more complex client-side build process, as Section 6.8 describes.

REST-less client apps are usually built using either a heavyweight JavaScript framework such as Vue, Ember, or ***React***. Less commonly, they can be built directly on top of the low-level UI abstractions provided by mobile OSes such as Android or iOS. REST-less client apps typically receive JSON data from the server and use it to manually construct or modify DOM elements. They may also receive static HTML from the server and attach JavaScript event handlers to the DOM on the fly to enable dynamic behavior, a process sometimes called ***hydration***. Some frameworks, such as React, can compile the same source code to either a JavaScript-based app or an OS-native app, but they accomplish this by abstracting away from HTML entirely, replacing it with functional-programming-like *render functions* and an HTML-like domain-specific language called *JSX*. As we will see in Section 6.8, this approach adds significant build-time complexity. Figure 6.15 shows the React implementation of the "movie review popover" developed in Figures 6.12 and 6.14 of Section 6.5.

RESTless app frameworks provide the ability to program graceful behavior while disconnected from the Internet, albeit using different mechanisms. Native apps offer maximal performance for sophisticated UX elements and can take full advantage of a mobile device's hardware capabilities, such as biometric authentication, easier native UX integration, and ***near-field communication*** sensors. But since native apps are manually installed by the user, they are reliant on the user for upgrades. The developer must deal with older or incompatible devices or OS versions, the inability to deprecate or remove older APIs for fear of stranding older app versions, and often the need to maintain multiple codebases for different platforms.

In contrast, JavaScript-based REST-less clients, like regular Web pages, are essentially downloaded on demand. One way that JavaScript-based apps can approach the functionality of native apps is to follow the design pattern of a Progressive Web App, or PWA (Russell 2015). Compared to a conventional RESTful client app, PWAs include two additional features that browsers must recognize and support. First, a PWA must include a ***manifest file*** that lists all assets required for the app to run—home screen icon, splash screen to display when the app is opened, other HTML pages, JavaScript code, CSS stylesheets, images, and so on—allowing these files to be downloaded and cached on the user's device, rather than fetched on demand every time the user activates the PWA. Second, when a PWA is activated it can ***spawn*** a *service worker*, essentially a JavaScript function that runs in a separate OS thread rather than in the browser window's main JavaScript thread, and can therefore continue running even when the browser window containing the PWA is not in the foreground. Service workers can *transparently* intercept `fetch()` requests when disconnected from the network and respond differently if the network is disconnected; the app's AJAX requests are oblivious to whether they will be handled by the server or by the service worker, but can be designed to gracefully handle a "network unavailable" response and take appropriate UX actions. Similarly, the service worker can subscribe to push notifications, that is, receive asynchronous messages from the server and "wake up" the main app. Since HTTP is a client-initiated request-reply protocol (Section 3.2), a server must use a push service to send

The service worker's ability to intercept HTTP requests is an example of the Proxy design pattern (Section 11.6).

	Task-oriented	**Content-oriented**
RESTful server-centric	Best practice ✓	Reduced complexity Better loading performance
REST-less JavaScript-based	Polyfill better UX Hide browser inconsistencies	Best practice ✓
REST-less native app		

Figure 6.16: Recommended/best practices for choosing the best client application architecture for the two different types of client apps. In an ideal world, web apps would fall into only two of these squares, but due to the limitations discussed and the need to satisfy critical business needs, they often choose otherwise. Note that I also filled in a fourth option not previously discussed—a content-oriented app that is fully RESTful—which is of course totally possible, and can have lower complexity and better loading performance.

such messages. The mechanics of doing so are beyond the scope of this brief exposition, but suffice to say that PWAs loosen the restriction of JavaScript client apps being strictly limited to the request-first constraint of HTTP.

Many businesses have both a web and native app version of their product, and native apps are built on different technology than the web, and have different constaints. Perhaps the most important difference from our point of view is that a native app is always *installed* on the user's device before use, which often includes downloading all of the code and data necessary to start using the app. For this reason, there is not much of a performance cost to client-side rendering in a native app, and in fact built-in native apps rendering APIs on platforms like iOS and Android do not support server-side rendering as a concept, because they are not based on a REST architecture.

If a business needs a native app, their server likely provides a JSON API, making frameworks like React more attractive to developers at these businesses, because they can then create a web app in a similar style to the native app and potentially re-use more code.

In practice, some native apps are simply installed wrappers around a *web view*, which is an API available on both iOS and Android that allows a UI within a native app to be drawn by a web page—it's basically a browser within an app. And there are many native apps that take this approach, in particular task-oriented apps often do so. That's because in such cases, there is not a significant UX advantage to going "full native". (As one example, the Amazon Android mobile app is mostly a thin wrapper around a web view.) In addition, Android and iOS offer various degrees of *installed web apps* (also known as PWAs), which are even thinner wrappers around a web app that don't require any app stores. And a web view-wrapper app or installed web app can easily be fully RESTful.

Our view is that task-oriented SaaS app use cases are most effectively served by a web app plus a wrapper web view if the business needs that. Alternatively, if there is a non-web view native app, the server can without too much effort expose both HTML and JSON variants of each route. For content-oriented apps, it's often worth the effort to build a full separate native app anyway, for the same reason it's worth it to build a fully client-side rendered web app.

■ ***Elaboration: HATEOAS***

why HATEOAS may obviate need to have a JS-based API to your server app

	Rspec	**Jasmine**	**Jest**	**Mocha**
Designed for	Plain Ruby apps, but `rspec-rails` provides Rails-specific conveniences; tests run in Rack application server	No assumptions about any JavaScript framwork; tests run in browser	React/Next.js and other "UI-heavy" apps; can work with other frameworks; comes with its own test runner app	Node.js apps; tests run in Node
Assertions	Ships with `rspec-expectations`, but can also use Shoulda or other assertion libraries with convenience methods	Built in	Built in	Requires an assertion library such as chai, Should.js, Express.js, etc.
Test doubles	Ships with `rspec-mocks`; can add FactoryBot for factories	Built in	Built in ???	

Figure 6.17: Comparison of test-writing tools for Ruby (left column) and JavaScript (right 3 columns). Some testing frameworks such as Jasmine include basic functionality for assertions and test doubles, while others such as Mocha require external packages for these functions.

6.7 Testing Client-Side Code

Surely, by this point in the book, the authors' views on the importance of testing are well established. The strategies in Chapter 8 extend readily to testing a server-side JavaScript app or testing a "full-framework" client-side app such as Google Docs; in keeping with the focus of this chapter, then, we turn our attention to the specific challenges of testing in-browser JavaScript in a server-centric, "task-oriented" app.

In some respects, this task is easier than either of the other two scenarios. For example, some happy paths, and some straightforward sad paths, can be covered by existing Cucumber/Capybara full-stack scenarios. By prepending the `@javascript` tag to a scenario, Cucumber will run that scenario in a ***headless browser*** that actually executes JavaScript, including making AJAX calls to the server, applying changes to the DOM, and so forth. In other words, the JavaScript is exercised as part of a regular full-stack test.

In other respects, testing in-browser JavaScript is more difficult. First, if such code inspects or modifies the DOM on specific pages, it probably expects some elements to be present or absent, to have certain content, or both; JavaScript test frameworks must provide a way to set these up, analogously to how fixtures and factories (Section 8.6) are used to set up the data structures and relationships that Rails models often expect to find when tests are run.

Second, whereas server errors tend to be noisy—raise an exception, write something useful to the logs, and so on—client-side errors are "silent" and result only in incorrect behavior (or no observable behavior at all), making bugs harder to detect.

Third, whereas SaaS server code needs to be tested only on the single technology stack hosting the server app (Section 1.6), variation among different browsers, browser versions, and client OSes increases the range of testing necessary to achieve comparable confidence in code correctness.

Figure 6.17 compares some popular JavaScript testing tools to the RSpec framework that

Chapter 8 uses for testing server code. Many properties of Jasmine make it ideal for our use here. It is among the most mature and widely used JavaScript testing framework. It is streamlined and quick to set up, and more than adequate for testing simple client-side code lacking "heavyweight" UI logic, as we would expect for server-centric "task-driven" apps. Jasmine makes no assumptions about client-side JavaScript frameworks, browsers, or DOM behaviors. Finally, its syntax and design support both Behavior-Driven Design (Chapter 7) and Test-Driven Development (Chapter 8) well.

In task-focused server-centric apps, the client code performs little or no business logic, so we expect few client-side tests that focus on it. Rather, we would expect many tests to be of the form "When a certain event occurs, does the correct behavior result?" Figure **??** shows a few kinds of events that might occur, the assertions we might want to check as a result, and some representative strategies for testing them. The key is to notice that for complex interactions such as AJAX calls, analogous to the discussion of "stubbing the Internet" in testing server-side code (Section 8.4), we can separate AJAX testing into at least two test cases. The first test case checks that when a certain event happens, the client-side code *tries to make* an AJAX (`fetch`) call to the server, and perhaps that the call's route and parameters are correct. The call itself will be stubbed out so that it does not actually happen during testing runs. The second test case checks that if a response arrives from the server for that call, the right things happen as a result of receiving that response. We can test this by simply constructing the payload of the server's assumed response and directly calling the correct callback function. This is the approach we will take in testing the AJAX examples from Section 6.5.

Of course, in production, the JavaScript code we are testing will run in a browser environment with specific HTML pages loaded, so its behavior may depend on the presence of particular elements in the DOM. We will show how to provide "test doubles" for these elements, analogous to using stubs and mocks (Section 8.3 and creating objects using factories (Section 8.6 when testing Ruby server code.

server side code: is the correct external API about to be called w/the right args? client: is the right AJAX call about to be made, eg fetch to correct URL, correct params/headers/etc?

server: allow 'canned' response to replace real API call. client: supply 'canned' response from server to AJAX call

Intro to jasmine very similar to Rspec, both in how test cases are expressed (anonymous lambdas) and the kinds of assertions available (give summary table; Jasmine in practice, example with HTMX? HTMX example like this (*), where you build a test document, mock out calls to the server, and then test that a certain interaction works. (this example uses chai for asseritons, Mocha, and Sinon; we can find/make an example using Jasmine and some reasonable HTMX library, showing the stubbing of AJAX calls as above)

TBD: These tables need updating, if nothign else to use the arrow function syntax

TBD: Talk about running Jasmine headless in CI

Structure of test cases

- `it("does something", function() {...})`
  Specifies a single test (spec) by giving a descriptive name and a function that performs the test.
- `describe("behaviors", function(){...})`
  Collects a related set of specs; the function body consists of calls to `it`, `beforeEach`, and `afterEach`. `describe`s can be nested.
- `beforeEach` and `afterEach`
  Setup/teardown functions that are run before each `it` block within the same `describe` block.

As with RSpec, if `describe`s are nested, all `beforeEach` are run from the outside in, and all `afterEach` from the inside out.

Expectations

An expectation in a spec takes the form `expect(`*object*`).`*expectation* or `expect(`*object*`).not.`*expectation*
Commonly used expectations built into Jasmine:

- `toEqual(`*val*`), toBeTruthy(), toBeFalsy()`
  Test for equality using `==`, or that an expression evaluates to Boolean true or false.

Commonly used expectations provided by the Jasmine jQuery add-on—in this case, the argument of `expect` should be a jQuery-wrapped element or set of elements:

- `toBeSelected(), toBeChecked(), toBeDisabled(), toHaveValue(stringValue)`
  Expectations on input elements in forms.
- `toBeVisible(), toBeHidden()`
  Hidden is true if the element has zero width and height, if it is a form input with `type="hidden"`, or if the element or one of its ancestors has the CSS property `display: none`.
- `toExist(), toHaveClass(`*class*`), toHaveId(`*id*`), toHaveAttr(`*attrName,attrValue*`)`
  Tests various attributes and characteristics of an element.
- `toHaveText(`*stringOrRegexp*`), toContainText(`*string*`)`
  Tests if the element's text exactly matches the given string or regexp, or contains the given substring.

Stubs (Spies)

- `spyOn(`*obj*`, 'func')`
  Creates and returns a spy (mock) of an existing function, which must be a function-valued property of *obj* named by `func`. The spy *replaces* the existing function.
- `calls` is a property of a spy that tracks calls that have been made to it, and the array `args[]` of the arguments of each call.

The following modifiers can be called on a spy to control its behavior:

- `and.returnValue(`*value*`)`
- `and.throwError(exception)`
- `and.callThrough()`
- `and.callFake(`*func*`)`
  `func` must be a function of zero arguments, though it has access to the arguments with which the spy was called via *spy*`.calls.mostRecent().args[]`, and can call other functions using these arguments.

Fixtures and factories (requires `jasmine-jquery`)

- `sandbox({class: 'myClass', id: 'myId'})`
  Creates an empty `div` with the given HTML attributes, if any; default is an empty `div` with no CSS class and an ID of `sandbox`. An alternative way to create the argument to `setFixtures` that avoids putting literal HTML strings into your test code.
- `loadFixtures("`*file*`.html")`
  Load HTML content from in `spec/javascripts/fixtures/`*file*`.html` and put it inside a `div` with ID `jasmine-fixtures`, which is cleaned out between test cases.

- `setFixtures(`*HTMLcontent*`)`
  Create a fixture directly instead of loading it from a file. *HTMLcontent* can be a literal string of HTML such as `<p class="foo">text</p>` or a jQuery-wrapped element such as `$('<p class="foo">text</p>')`.
- `getJSONFixture("`*file*`.json")`
  Returns the JSON object in `spec/javascripts/fixtures/`*file*`.json`. Useful for storing mock data to simulate the result of an AJAX call without having to put literal JSON objects into your test code.

RSpec/Ruby	Jasmine/JavaScript
obj.mock('method') and_return(value) and_raise(exception) expect expr .	jasmine.spyOn(obj, 'method') andReturn(value) andThrow(exception) andCallFake(function) andCallThrough()

What	RSpec/Ruby	Jasmine/JavaScript
Libraries	`rspec, rspec-rails` gems	`jasmine` gem, `jasmine-jquery` add-on
Setup	`rails generate rspec:install`	`rails generate jasmine:install`
Test files	`spec/models/, spec/controllers/, spec/helpers`	`spec/javascripts/`
Naming conventions	`spec/models/movie_spec.rb` contains tests for `app/models/movie.rb`	`spec/javascripts/movie_popup_spec.js` contains tests for `app/assets/javascripts/movie_popup.js`; `spec/javascripts/moviePopupSpec.js` contains tests for `app/assets/javascripts/moviePopup.js`
Configuration file	`.rspec`	`spec/javascripts/support/jasmine.yml`
Run all tests	`rake spec`	`rake jasmine`, then visit `http://localhost:8888`; or `rake jasmine:ci` to run once using Selenium/Webdriver and capture the output; or use `jasmine-headless-webkit`[11] to run from command line with no browser

TBD: replace jasmine-jquery expectations with some that work with plain JSAPI

Finally, good unit-testing discipline requires us to be able to test the client-side JavaScript code without calling the server every time. Recall from Section 8.4 how stubbing the Internet to isolate tests from external services can be done either "near the client" or "far from the client." In Section 6.7 we stubbed "near the client" by stubbing `$.ajax` and forcing it to immediately call the `success` function rather than allowing it to proceed with the external HTTP request. Another way to stub near the client for SPAs is Jasmine-jQuery's fixture mechanism, which allows us to specify JSON fixtures as well as HTML fixtures, as Figure **??** shows. You can make a call to the actual server, capture the response as a JSON fixture, and use the fixture as a "canned" response in Jasmine tests, similar to how we stubbed `find_in_tmdb` in Section 8.4 to return a value immediately rather than allowing it to make a real HTTP request.

An alternative, which would more thoroughly exercise the code that handles the actual AJAX server responses, is to stub at the network level. Just as Webmock lets you provide "canned" responses based on the arguments of an XHR call, `jasmine-ajax`[12], a Jasmine extension from Pivotal Labs, lets you provide "canned" XML, HTML or JSON responses to AJAX XHR calls that are used instead of allowing the XHR call to proceed. You can then spy on the handler functions `success`, `failure`, `timeout`,

https://gist.github.com/u/saasbook

```
1 // TBD javascript example
```

Figure 6.18: In a JavaScript module, a class, variable, or function name can be preceded by `export`, or one or more curly-braced lists of names to export can appear anywhere in the module. A module that wants to import names exported by another module can either import these names as provided by the exporting module, or if there are name collisions, can rename the names as they are imported.

and so on passed to `$.ajax` to make sure the correct handler is called depending on the server's response.

6.8 Build Tools For Client-Side Programming

The more libraries and code a web app client has, the more the problems of scale become apparent: large code bases start to become ungainly, more prone to bugs, and less performant. There are software engineering approaches to address each of them, but they generally all involve using build-time and serving-time infrastructure tools. For example:

- Large amounts of JavaScript can be split into different *JavaScript modules*.
- Better yet, writing code at all can be avoided by including third-party JavaScript libraries for common tasks. A huge number of them are available on repositories like *NPM* (Node Package Manager).
- The risk of bugs can be mitigated with approaches like adding typing to JavaScript with JavaScript extension languages such as *TypeScript*.
- The performance cost of lots of JavaScript (as well as the cost of lots of JavaScript modules) can be mitigated with libraries like *Vite* that help to deliver only the JavaScript needed for each page, and in an optimized form.

Prior to ES6, many *ad hoc* systems external to the language were used to manage JavaScript namespaces. Some aspects of the ES6 module syntax are aimed at maximizing backwards compatibility with these schemes, notably ***CommonJS*** and ***Asynchronous Module Definition***. The major distinction is whether a given file when loaded is treated as a *script* or (preferred) as a *module*. In a script, all variables and functions declared at top level become visible in the global scope. In a module, only variables and functions specifically marked with the keyword `export` are visible outside the module; everything else is scoped to the module, as Figure 6.19 shows.

JavaScript modules are already understood by all web browsers, so unlike TypeScript (which we describe below) you can simply load every module while developing your web app. When *deploying*, however, especially if you have lots of modules, simply loading every module can cause performance problems (one HTTP request per module), or they depend on each other transitively (the waterfall problem mentioned earlier in this chapter). Tools like Vite **TBD:** ref or URL? can be included in your build to add a step that reads all your JavaScript modules, elminates dead (not callable) code, and *bundles* the remaining contents into a smaller number of larger JavaScript files, thus avoiding extra HTTP overhead and waterfalls. For example, if a page's HTML directly loads JavaScript modules A and B, and A depends on C, and C on D, then Vite may concatenate D, C, A, and B (in that order) into one bundle; the HTML can then be changed to load that bundle directly instead of A and B individually.

Tree shaking is how the JavaScript community refers to dead code elimination.

Third-party libraries can also be imported via JavaScript module syntax, but incur additional build steps to import those libraries into local directories, build them (they may have their own internal build steps), and keep them up to date as those dependencies change over time. Furthermore, it's quite common for such dependencies to have dependencies of their own, leading to even more complexity and build steps.

The HTTP overhead avoided by bundling is decreasing over time due to new technologies like HTTP/2 that allow multiplexing multiple requests on one connection. Furthermore, bundling harms caching when only some modules change in the future. On the other

Addressing concerns beyond the quantity of code and number of classes, TypeScript is a very popular way of augmenting JavaScript to add *optional types*. It can detect errors such as trying to access properties on objects that were never defined or referencing undefined variables (easy mistakes to make in JavaScript just by misspelling a word), or passing the wrong object type to methods. In other words, it offers many of the advantages of strongly typed languages like C++, but for JavaScript.

TODO: quick example

Unfortunately, since browsers don't understand TypeScript (and even if they did, you'd want to find errors automatically before trying to load the page in a web browser), adopting TypeScript requires a build step that compiles TypeScript into JavaScript by checking and then removing the types, resulting in plain JavaScript.

All these tools and libraries can be great for productivity, quality and performance, but they come at a cost of more and more build steps, and more and more complexity generally:

- A slower build harms productivity (TODO: cite number of seconds before distraction).
- More complexity can reduce productivity and quality.
- Performance optimization can only go so far to mitigate inherent app runtime cost.

If you choose React for your web app, you will also need build steps related to converting JSX to JavaScript; the library that does that is called Babel (likely named after the universal translation Babel Fish featured in *The Hitchhiker's Guide to the Galaxy*). The React examples in this chapter use a JavaScript implementatoin of Babel to convert at browser load time, but that is generally too slow for a real-world web app.

So in the end, as with everything in life, there are inherent tradeoffs to all of them, and the net impact on your web app may be positive or negative depending on whether their costs are less than or greater than the problem they are solving.

Similar to the discussion earlier in the chapter, the simpler the needs of the web app, or the more task-oriented it is, the less likely there is a need for lots of software engineering tools. For example, an elegant solution to "my task-oriented web app loads too slowly due to a JavaScript waterfall or size bloat" is often "find a way to use less JavaScript and rely more on the browser", since doing so successfully will often improve page load times, reduce complexity and avoid maintenance burden. On the other hand, a complex and immersive web app may have a whole lot of functionality that is inherently hard to get right, and so justifies using a battery of software engineering tools like type checkers and so on.

Armando says: one practical piece he can contribute here is how Rails helper methods like `javascript_include_tag` work differently in development (where they arrange to reload every single .js file on every requet, to get hot changes) vs production (where they are usually configured to trigger running the asset builder).

■ ***Elaboration: Other JavaScript-like languages***

Since JavaScript is the web's only scripting language, developers have designed different languages that have nicer (in their opinion) syntax, provide syntactic sugar for features that plain JavaScript lacks, or avoid awkward parts of JavaScript. Each such language requires a compiler or ***transpiler***) to convert it to plain JavaScript so it can actually run in a browser (or in a server-side system such as Node). An example of such a language is TypeScript, which adds static typing and optional type annotations to JavaScript. The TypeScript compiler is both written in JavaScript and uses it as the target language. Another example is the transpiled language CoffeeScript[13], which does not offer typing but provides a less verbose syntax that some developers prefer over regular JavaScript. Notwithstanding, all of these alternative languages must essentially be syntactic sugar alternatives to JavaScript. Although these languages arise from developers' desire to become more productive, transpilers and compilers slow down build times and can complicate debugging. As a result, there is an effort underway to split JavaScript into a core language called *JS0*, which all browsers would support, and *JSsugar*, which would add ergonomic features similar to what languages like CoffeeScript provide. JSSugar will still require an additional compilation step, but because it'll be standardized, it should maintain compatibility across more libraries and tools than proprietary alternatives.

6.9 Fallacies and Pitfalls

***Fallacy:* Client-side apps require frameworks such as React or React Native to feel smooth and professional**

As Section 6.6 described, most task-oriented apps and some content-oriented apps today (2025) can achieve a level of "appinness" that would have required such frameworks 5 or 10 years ago. Since choosing a heavyweight framework over JavaScript-augmented server-centric apps comes with nontrivial costs in development time, code maintenance, and keeping apps up-to-date, your authors believe you need a good reason to use one, rather than making it the default choice.

***Pitfall:* "JSON for everything" leading to non-DRY code**

As Section 6.5 described, receiving JSON from the server may result in client-side JavaScript code having to render HTML itself in order to modify the DOM, essentially duplicating the view-rendering logic that already exists on the server. Writing rendering logic twice in two languages seems wasteful—it violates DRY, reinvents a working solution (server-side rendering), wastes programmer time, and perhaps also makes things more complicated than they need to be. And duplication of logic can lead to bugs arising from differences between the server and client code.

Web app developers have come up with three solutions for this duplication problem.

1. Do all the rendering on the client instead of the server. In this approach, the server returns only stub HTML that bootloads some JavaScript to do the actual rendering, and then that JavaScript requests JSON from the server and renders it into HTML (or, perhaps the stub HTML also contains some JSON inlined into it to avoid more server requests). This approach avoids duplication of code, but it leads to a lot more JavaScript on the client, and significantly slower page loads as a result of loading all that JavaScript and using it to render.
2. Write the server in the same language as the client. In this approach, JavaScript would be used instead of Ruby to render on the server. Typically the server would use *node.js* to run JavaScript with a special DOM shim that knows how to serialize the result to a string. This approach also avoids duplication, but loses all of the power, convenience and language features not in JavaScript of Rails (or any other server-specific rendering framework). It also has much of the performance problems of the first option, since all client renders need the template code; the difference is that the first render of a web app load renders HTML much more quickly if it is pre-rendered server-side.
3. Don't return JSON from the server if it's intended for UI—return HTML instead. This has optimal loading performance and avoids code duplication, but it means client-side interactions now *require* requesting updates from the server, which could be slower if the data was otherwise available client-side.

The potentially improved UX of "client-side rendering updates without waiting for a server request" is the main the reason to prefer one of the first two approaches. Stateful and long-lived web apps with complex UIs that must be very responsive or work offline—like Google Docs, Figma, or Gmail—likely justify client-side rendering. Server-centric apps used in short transactional bursts may not.

***Fallacy:* HTTP/2 solves all problems with resource loading**

Section 6.2 introduced JavaScript modules—a convenient way to split up code into logical chunks. For non-web programming, you can modularize your code in this way without a second thought. However, by default modularized JavaScript loaded into a web app performs one HTTP request for each module. This could be expensive for two reasons:

- *Parallel overhead*: sending many requests at once clogs up the network, CPU and cache.

- *Serial overhead*: a chain of JavaScript module dependencies leads to multiple network requests in sequence, wasting time waiting for information to come and go over the network repeatedly.

Most parallel overhead can be avoided with HTTP/2, because this version of HTTP supports multiple requests over one HTTP connection.

The serial overhead problem is often called a "waterfall", because if you plot the duration of each request in a line chart that has time on the horizontal axis and sequence of requests on the vertical, it looks like a waterfall cascading downward and to the right.

Serial overhead can be avoided by adding a *bundling* build step that pre-computes the list of modules you'll need to load for each page and puts them all in a single script; this script is then referenced in the HTML page at runtime rather than the individual modules. Doing this may reduce parallel overhead as well.

***Pitfall:* Using async/await without understanding how it works**

TBD: Pitfall example: something that can go wrong when blindly using async/await syntax without an understanding of what's happening behind the scenes.

***Pitfall:* Truthiness is handled differently in different languages**

Be careful when using non-Boolean expressions as conditions and on the use of special values. For example, whereas Ruby uses `nil` to mean both "undefined" (a variable that has never been given a value) and "empty" (a value that is always false), JavaScript's `null` is distinct from its `undefined`, which is what you get as the "value" of a variable that has never been initialized. Similarly, in Ruby every expression other than `nil` or `false` is truthy, while in JavaScript (as Figure 6.2 shows) many non-Boolean expressions are falsy.

6.10 Concluding Remarks: JavaScript Past, Present and Future

TBD: Discussion about 'spectrum of frameworks'

Mention other frameworks like jquery, bootstrap, styles.js, etc and how they fit on the spectrum, maybe even have a linear representation of spectrum with these points on it.

As the preceding sections have argued, for task-oriented apps, often the most elegant way to make the app streamlined and performant is to avoid excessive use of JavaScript and rely more on the browser, a complex piece of software in which millions of engineer-hours have been invested over the years. Doing so successfully will often improve page load times, reduce complexity and avoid maintenance burden.

At one end of the spectrum is HATEOAS ...

What is HATEOAS and why it may obviate the need for JS logic

At the other end of the spectrum are complex and immersive web apps whose functionality is inherently hard to get right, and so justifies using a battery of additional software engineering tools. This is where heavyweight front-end frameworks come in. Some are based on standard JavaScript (Vue), others on JavaScript variants such as TypeScript (**TBD:** example?), yet others on an entirely different source representation (React). Whichever the case, as a web app client grows to encompass more front-end code, the codebase starts to become ungainly, more prone to bugs, and less performant. There are software engineering approaches to address each of them, but they generally all involve using build-time and serving-time infrastructure tools.

At the simplest level, JavaScript code can be split into different *modules* (Figure 6.19), which are supported as of ES6. At serving time, the browser will load each JavaScript file as either a *script* or (preferred) as a *module*. In a script, all variables and functions declared at top level become visible in the global scope; in a module, only variables and functions specifically marked with the keyword `export` are visible outside the module.

https://gist.github.com/u/saasbook

```
1 // TBD javascript example
```

Figure 6.19: In a JavaScript module, a class, variable, or function name can be preceded by `export`, or one or more curly-braced lists of names to export can appear anywhere in the module. A module that wants to import names exported by another module can either import these names as provided by the exporting module, or if there are name collisions, can rename the names as they are imported.

In production, however, managing lots of modules takes some care. If you have lots of modules, simply loading all of them may cause performance problems, since each module requires one HTTP request. ***HTTP/2*** mitigates this specific problem, but If modules depend on each other, they may have to be loaded in a particular order **TBD:** is this true?. Tools such as Vite **TBD:** ref or URL? can be included in your build to add a step that reads all your JavaScript modules, eliminates dead (not callable) code, and *bundles* the remaining contents into a smaller number of larger JavaScript files, thus avoiding extra HTTP overhead and waterfalls. For example, if a page's HTML directly loads JavaScript modules A and B, and A depends on C, and C on D, then Vite may concatenate D, C, A, and B (in that order) into one bundle; the HTML can then be changed to load that bundle directly instead of A and B individually.

Tree shaking is how the JavaScript community refers to dead code elimination.

If you use React, you will also need build steps related to converting JSX to JavaScript, using the Babel library. Although there is a JavaScript implementation of Babel that does this conversion at browser load time, that approach is generally too slow for a real-world web app, requiring the addition of build steps instead.

Likely named after the universal translation BabelFish featured in *The Hitchhiker's Guide to the Galaxy*.

We reiterate that many of the JavaScript tasks that these frameworks abstract away—fetching a resource from a server, error-checking the response status, parsing the response body, possibly modifying the DOM using the body contents, and so on—*are the exact tasks done by browsers themselves.* As browsers have embraced the latest HTML5 and CSS3 features, behaviors that previously required JavaScript can now be achieved entirely declaratively, allowing removal of code—which often leads to fewer bugs.

In fairness, simpler and earlier frameworks seem to have long-tailed lifetimes, especially when they are used by popular software: according to W3techs[14], jQuery is still used on about 75% of all websites they scraped, and much of that usage is due to its being part of the popular WordPress blog software. But in the opinion of your authors, a project needs a specific reason to use a "heavyweight" JavaScript framework, rather than a reason not to.

Addressing concerns beyond the quantity of code and number of classes, developers have designed different languages that have nicer (in their opinion) syntax than JavaScript, provide syntactic sugar for features that plain JavaScript lacks, or avoid awkward parts of JavaScript. Since JavaScript is the web's only scripting language, each "enhanced" language requires a compiler or ***transpiler***) to convert it to plain JavaScript so it can actually run in a browser. As with React, in practice this transpiling must happen at build time. One such language is TypeScript, which adds static typing and optional type annotations to JavaScript. TypeScript can detect errors such as trying to access properties on objects that were never defined, referencing undefined variables (easy to do in JavaScript just by misspelling a word), or passing the wrong object type to methods. In other words, it offers many of the advantages of strongly typed languages like C++, but for JavaScript. Although such languages arise from developers' desire to become more productive, transpilers and compilers slow down build times and can complicate debugging. As a result, there is an effort underway to split JavaScript into a core language called *JS0*, which all browsers would support, and *JSsugar*, which would add ergonomic features similar to what languages like CoffeeScript provide. JSSugar will still require an additional compilation step, but because it'll be standardized, it should maintain compatibility across more libraries and tools than proprietary alternatives.

All these tools and libraries can be great for productivity, quality and performance, but they come at a cost of more and more build steps, and more and more complexity generally. Slower builds hurt

productivity. Added complexity can reduce both productivity and quality. And performance optimizations such as those done by Vite can only go so far to mitigate the inherent runtime cost of just adding a lot of JavaScript code.

So in the end, as with everything in software engineering (and life), there are inherent tradeoffs to all of them, and the net impact on your web app may be positive or negative depending on whether their costs are less than or greater than the problem they are solving. The decision to have a heavyweight front end should be made with care. Even if you do decide to go that route, given the churn and evolutionary speed of the front-end ecosystem, a developer who knows a particular framework but is unfamiliar with the basic operations that it abstracts away will face difficulties when that framework goes out of vogue. Frameworks or not, there is still no substitute for conceptual mastery of the basic architectural concepts underlying the server-centric Web.

TBD: where to put this - an example of JS evolution For example, the `fetch` API can be polyfilled by writing some code that defines the API and implements it using `XMLHttpRequest` and the `Promise` API; this is an example of using JavaScript to polyfill another JavaScript API. Web apps often include polyfils for JavaScript APIs present in only some of the popular browsers, so that the web app can use that API in the meantime.

A. Russell, Aug. 2015. URL `https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955`.

P. Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009. ISBN 1430219483.

Notes

[1] `http://github.com/jasmine/jasmine`
[2] `https://html.spec.whatwg.org/`
[3] `https://www.nngroup.com/articles/original-top-ten-mistakes-in-web-design/`
[4] `https://erlangforums.com/t/heroku-replaces-its-erlang-based-router-with-go/3826`
[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript`
[6] `jquery.org`
[7] `getbootstrap.com`
[8] `https://htmx.org`
[9] `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_transitions/Using_CSS_transitions`
[10] `https://developer.mozilla.org/en-US/docs/Web/API/View_Transition_API`
[11] `http://johnbintz.github.com/jasmine-headless-webkit/`
[12] `https://github.com/jasmine/jasmine-ajax`
[13] `https://coffeescript.org`
[14] `w3techs.com`