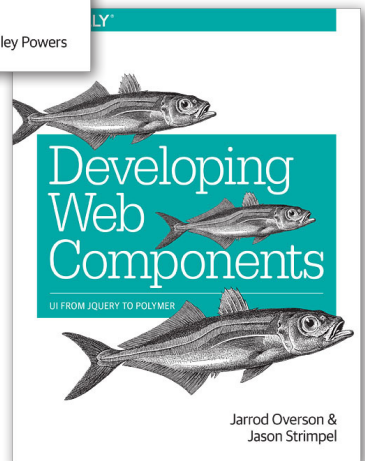
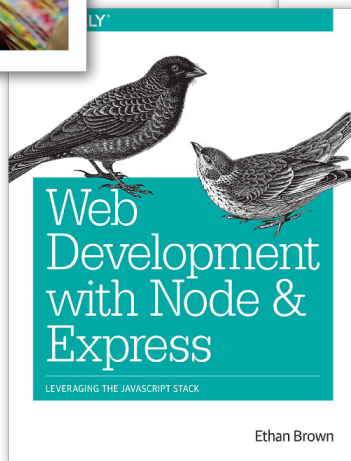
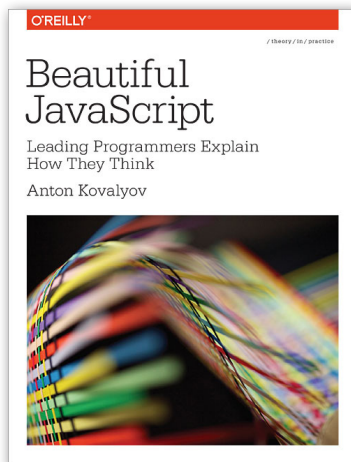


Modern JavaScript

A Curated Collection of Chapters
from the O'Reilly JavaScript Library

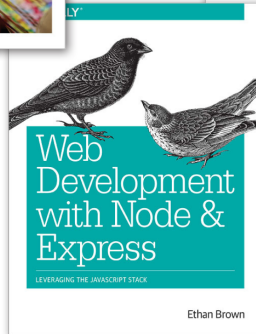
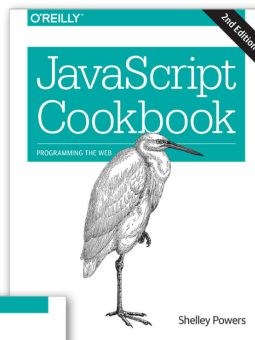
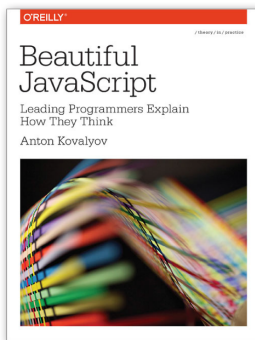


Modern JavaScript

A Curated Collection of Chapters from the O'Reilly JavaScript Library

JavaScript has come a long way. It may have seemed like a “toy language” at first, but it has evolved into the powerful dominant scripting language of the Web. JavaScript is now found not only in the browser, but on the server, and it’s even moving into the world of hardware. Staying on top of the latest methodologies, tools, and techniques is critical for any JavaScript developer, whether you’re building single-page web apps with front-end frameworks or building a RESTful API in Node.js.

This free ebook gets you started, bringing together concepts that you need to understand before tackling your next modern JavaScript app. With a collection of chapters from the O'Reilly JavaScript library's published and forthcoming books, you'll learn about the scope and challenges that await you in the world of modern web development.



Web Development with Node & Express

[Available here](#)

Chapter 1: Introducing Express

Chapter 2: Getting Started with Node

JavaScript Cookbook, Second Edition

[Available here](#)

Chapter 12: Modularizing and Managing JavaScript

Developing Web Components

[Available here](#)

Chapter 11: Working with the Shadow DOM

Beautiful JavaScript

[Available here](#)

Chapter 14: Functional JavaScript



Web Development with Node & Express

LEVERAGING THE JAVASCRIPT STACK

Ethan Brown

Web Development with Node and Express

Ethan Brown

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Introducing Express

The JavaScript Revolution

Before I introduce the main subject of this book, it is important to provide a little background and historical context, and that means talking about JavaScript and Node.

The age of JavaScript is truly upon us. From its humble beginnings as a client-side scripting language, not only has it become completely ubiquitous on the client side, but its use as a server-side language has finally taken off too, thanks to Node.

The promise of an all-JavaScript technology stack is clear: no more context switching! No longer do you have to switch mental gears from JavaScript to PHP, C#, Ruby, or Python (or any other server-side language). Furthermore, it empowers frontend engineers to make the jump to server-side programming. This is not to say that server-side programming is strictly about the language: there's still a lot to learn. With JavaScript, though, at least the language won't be a barrier.

This book is for all those who see the promise of the JavaScript technology stack. Perhaps you are a frontend engineer looking to extend your experience into backend development. Perhaps you're an experienced backend developer like myself who is looking to JavaScript as a viable alternative to entrenched server-side languages.

If you've been a software engineer for as long as I have, you have seen many languages, frameworks, and APIs come into vogue. Some have taken off, and some have faded into obsolescence. You probably take pride in your ability to rapidly learn new languages, new systems. Every new language you come across feels a little more familiar: you recognize a bit here from a language you learned in college, a bit there from that job you had a few years ago. It feels good to have that kind of perspective, certainly, but it's also wearying. Sometimes you want to just *get something done*, without having to learn a whole new technology or dust off skills you haven't used in months or years.

JavaScript may seem, at first, an unlikely champion. I sympathize, believe me. If you told me three years ago that I would not only come to think of JavaScript as my language of choice, but also write a book about it, I would have told you you were crazy. I had all the usual prejudices against JavaScript: I thought it was a “toy” language. Something for amateurs and dilettantes to mangle and abuse. To be fair, JavaScript did lower the bar for amateurs, and there was a lot of questionable JavaScript out there, which did not help the language’s reputation. To turn a popular saying on its head, “Hate the player, not the game.”

It is unfortunate that people suffer this prejudice against JavaScript: it has prevented people from discovering how powerful, flexible, and elegant the language is. Many people are just now starting to take JavaScript seriously, even though the language as we know it now has been around since 1996 (although many of its more attractive features were added in 2005).

By picking up this book, you are probably free of that prejudice: either because, like me, you have gotten past it, or because you never had it in the first place. In either case, you are fortunate, and I look forward to introducing you to Express, a technology made possible by a delightful and surprising language.

In 2009, years after people had started to realize the power and expressiveness of JavaScript as a browser scripting language, Ryan Dahl saw JavaScript’s potential as a server-side language, and Node was born. This was a fertile time for Internet technology. Ruby (and Ruby on Rails) took some great ideas from academic computer science, combined them with some new ideas of its own, and showed the world a quicker way to build websites and web applications. Microsoft, in a valiant effort to become relevant in the Internet age, did amazing things with .NET and learned not only from Ruby and JavaScript, but also from Java’s mistakes, while borrowing heavily from the halls of academia.

It is an exciting time to be involved in Internet technology. Everywhere, there are amazing new ideas (or amazing old ideas revitalized). The spirit of innovation and excitement is greater now than it has been in many years.

Introducing Express

The Express website describes Express as “a minimal and flexible node.js web application framework, providing a robust set of features for building single and multipage and hybrid web applications.” What does that really mean, though? Let’s break that description down:

Minimal

This is one of the most appealing aspects of Express. Many times, framework developers forget that usually “less is more.” The Express philosophy is to provide the *minimal* layer between your brain and the server. That doesn’t mean that it’s not

robust, or that it doesn't have enough useful features. It means that it gets in your way less, allowing you full expression of your ideas, while at the same time providing something useful.

Flexible

Another key aspect of the Express philosophy is that Express is extensible. Express provides you a very minimal framework, and you can add in different parts of Express functionality as needed, replacing whatever doesn't meet your needs. This is a breath of fresh air. So many frameworks give you *everything*, leaving you with a bloated, mysterious, and complex project before you've even written a single line of code. Very often, the first task is to waste time carving off unneeded functionality, or replacing the functionality that doesn't meet requirements. Express takes the opposite approach, allowing you to add what you need when you need it.

Web application framework

Here's where semantics starts to get tricky. What's a web application? Does that mean you can't build a website or web pages with Express? No, a website *is* a web application, and a web page *is* a web application. But a web application can be more: it can provide functionality to *other* web applications (among other things). In general, "app" is used to signify something that has functionality: it's not just a static collection of content (though that is a very simple example of a web app). While there is currently a distinction between an "app" (something that runs natively on your device) and a "web page" (something that is served to your device over the network), that distinction is getting blurrier, thanks to projects like PhoneGap, as well as Microsoft's move to allow HTML5 applications on the desktop, as if they were native applications. It's easy to imagine that in a few years, there won't be a distinction between an app and a website.

Single-page web applications

Single-page web applications are a relatively new idea. Instead of a website requiring a network request every time the user navigates to a different page, a single-page web application downloads the entire site (or a good chunk of it) to the client's browser. After that initial download, navigation is faster because there is little or no communication with the server. Single-page application development is facilitated by the use of popular frameworks such as Angular or Ember, which Express is happy to serve up.

Multipage and hybrid web applications

Multipage web applications are a more traditional approach to websites. Each page on a website is provided by a separate request to the server. Just because this approach is more traditional does not mean it is not without merit or that single-page applications are somehow better. There are simply more options now, and you can decide what parts of your content should be delivered as a single-page app, and

what parts should be delivered via individual requests. “Hybrid” describes sites that utilize both of these approaches.

If you’re still feeling confused about what Express actually *is*, don’t worry: sometimes it’s much easier to just start using something to understand what it is, and this book will get you started building web applications with Express.

A Brief History of Express

Express’s creator, TJ Holowaychuk, describes Express as a web framework inspired by Sinatra, which is a web framework based on Ruby. It is no surprise that Express borrows from a framework built on Ruby: Ruby spawned a wealth of great approaches to web development, aimed at making web development faster, more efficient, and more maintainable.

As much as Express was inspired by Sinatra, it is also deeply intertwined with Connect, a “plugin” library for Node. Connect coined the term “middleware” to describe pluggable Node modules that can handle web requests to varying degrees. Up until version 4.0, Express bundled Connect; in version 4.0, Connect (and all middleware except `static`) was removed to allow these middleware to be updated independently.



Express underwent a fairly substantial rewrite between 2.x and 3.0, then again between 3.x and 4.0. This book will focus on version 4.0.

Upgrading to Express 4.0

If you already have some experience with Express 3.0, you’ll be happy to learn that upgrading to Express 4.0 is pretty painless. If you’re new to Express, you can skip this section. Here are the high points for those with Express 3.0 experience:

- Connect has been removed from Express, so with the exception of the `static` middleware, you will need to install the appropriate packages (namely, `connect`). At the same time, Connect has been moving some of its middleware into their own packages, so you might have to do some searching on npm to figure out where your middleware went.
- `body-parser` is now its own package, which no longer includes the `multipart` middleware, closing a major security hole. It’s now safe to use the `body-parser` middleware.
- You no longer have to link the Express router into your application. So you should remove `app.use(app.router)` from your existing Express 3.0 apps.

- `app.configure` was removed; simply replace calls to this method by examining `app.get(env)` (using either a `switch` statement or `if` statements).

For more details, see the [official migration guide](#).

Express is an open source project and continues to be primarily developed and maintained by TJ Holowaychuk.

Node: A New Kind of Web Server

In a way, Node has a lot in common with other popular web servers, like Microsoft's Internet Information Services (IIS) or Apache. What is more interesting, though, is how it differs, so let's start there.

Much like Express, Node's approach to web servers is very minimal. Unlike IIS or Apache, which a person can spend many years mastering, Node is very easy to set up and configure. That is not to say that tuning Node servers for maximum performance in a production setting is a trivial matter: it's just that the configuration options are simpler and more straightforward.

Another major difference between Node and more traditional web servers is that Node is single threaded. At first blush, this may seem like a step backward. As it turns out, it is a stroke of genius. Single threading vastly simplifies the business of writing web apps, and if you need the performance of a multithreaded app, you can simply spin up more instances of Node, and you will effectively have the performance benefits of multithreading. The astute reader is probably thinking this sounds like smoke and mirrors. After all, isn't multithreading through server parallelism (as opposed to app parallelism) simply moving the complexity around, not eliminating it? Perhaps, but in my experience, it has moved the complexity to exactly where it should be. Furthermore, with the growing popularity of cloud computing and treating servers as generic commodities, this approach makes a lot more sense. IIS and Apache are powerful indeed, and they are designed to squeeze the very last drop of performance out of today's powerful hardware. That comes at a cost, though: they require considerable expertise to set up and tune to achieve that performance.

In terms of the way apps are written, Node apps have more in common with PHP or Ruby apps than .NET or Java apps. While the JavaScript engine that Node uses (Google's V8) does compile JavaScript to native machine code (much like C or C++), it does so transparently,¹ so from the user's perspective, it behaves like a purely interpreted language. Not having a separate compile step reduces maintenance and deployment hassles: all you have to do is update a JavaScript file, and your changes will automatically be available.

1. Often called "Just in Time" (JIT) compilation.

Another compelling benefit of Node apps is that Node is incredibly platform independent. It's not the first or only platform-independent server technology, but platform independence is really more of a spectrum than a binary proposition. For example, you can run .NET apps on a Linux server thanks to Mono, but it's a painful endeavor. Likewise, you can run PHP apps on a Windows server, but it is not generally as easy to set up as it is on a Linux machine. Node, on the other hand, is a snap to set up on all the major operating systems (Windows, OS X, and Linux) and enables easy collaboration. Among website design teams, a mix of PCs and Macs is quite common. Certain platforms, like .NET, introduce challenges for frontend developers and designers, who often use Macs, which has a huge impact on collaboration and efficiency. The idea of being able to spin up a functioning server on any operating system in a matter of minutes (or even seconds!) is a dream come true.

The Node Ecosystem

Node, of course, lies at the heart of the stack. It's the software that enables JavaScript to run on the server, uncoupled from a browser, which in turn allows frameworks written in JavaScript (like Express) to be used. Another important component is the database, which will be covered in more depth in [Chapter 13](#). All but the simplest of web apps will need a database, and there are databases that are more at home in the Node ecosystem than others.

It is unsurprising that database interfaces are available for all the major relational databases (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server): it would be foolish to neglect those established behemoths. However, the advent of Node development has revitalized a new approach to database storage: the so-called “NoSQL” databases. It's not always helpful to define something as what it's *not*, so we'll add that these NoSQL databases might be more properly called “document databases” or “key/value pair databases.” They provide a conceptually simpler approach to data storage. There are many, but MongoDB is one of the frontrunners, and the one we will be using in this book.

Because building a functional website depends on multiple pieces of technology, acronyms have been spawned to describe the “stack” that a website is built on. For example, the combination of Linux, Apache, MySQL, and PHP is referred to as the *LAMP* stack. Valeri Karpov, an engineer at MongoDB, coined the acronym *MEAN*: Mongo, Express, Angular, and Node. While it's certainly catchy, it is limiting: there are so many choices for databases and application frameworks that “MEAN” doesn't capture the diversity of the ecosystem (it also leaves out what I believe is an important component: templating engines).

Coining an inclusive acronym is an interesting exercise. The indispensable component, of course, is Node. While there are other server-side JavaScript containers, Node is emerging as the dominant one. Express, also, is not the only web app framework available, though it is close to Node in its dominance. The two other components that are

usually essential for web app development are a database server and a templating engine (a templating engine provides what PHP, JSP, or Razor provides naturally: the ability to seamlessly combine code and markup output). For these last two components, there aren't as many clear frontrunners, and this is where I believe it's a disservice to be restrictive.

What ties all these technologies together is JavaScript, so in an effort to be inclusive, I will be referring to the "JavaScript stack." For the purposes of this book, that means Node, Express, and MongoDB.

Licensing

When developing Node applications, you may find yourself having to pay more attention to licensing than you ever have before (I certainly have). One of the beauties of the Node ecosystem is the vast array of packages available to you. However, each of those packages carries its own licensing, and worse, each package may depend on other packages, meaning that understanding the licensing of the various parts of the app you've written can be tricky.

However, there is some good news. One of the most popular licenses for Node packages is the MIT license, which is painlessly permissive, allowing you to do *almost* anything you want, including use the package in closed source software. However, you shouldn't just assume every package you use is MIT licensed.



There are several packages available in npm that will try to figure out the licenses of each dependency in your project. Search npm for `license-sniffer` or `license-spelunker`.

While MIT is the most common license you will encounter, you may also see the following licenses:

GNU General Public License (GPL)

The GPL is a very popular open source license that has been cleverly crafted to keep software free. That means if you use GPL-licensed code in your project, your project must *also* be GPL licensed. Naturally, this means your project can't be closed source.

Apache 2.0

This license, like MIT, allows you to use a different license for your project, including a closed source license. You must, however, include notice of components that use the Apache 2.0 license.

Berkeley Software Distribution (BSD)

Similar to Apache, this license allows you to use whatever license you wish for your project, as long as you include notice of the BSD-licensed components.



Software is sometimes *dual licensed* (licensed under two different licenses). A very common reason for doing this is to allow the software to be used in both GPL projects and projects with more permissive licensing. (For a component to be used in GPL software, the component must be GPL licensed.) This is a licensing scheme I often employ with my own projects: dual licensing with GPL and MIT.

Lastly, if you find yourself writing your own packages, you should be a good citizen and pick a license for your package, and document it correctly. There is nothing more frustrating to a developer than using someone's package and having to dig around in the source to determine the licensing or, worse, find that it isn't licensed at all.

Getting Started with Node

If you don't have any experience with Node, this chapter is for you. Understanding Express and its usefulness requires a basic understanding of Node. If you already have experience building web apps with Node, feel free to skip this chapter. In this chapter, we will be building a very minimal web server with Node; in the next chapter, we will see how to do the same thing with Express.

Getting Node

Getting Node installed on your system couldn't be easier. The Node team has gone to great lengths to make sure the installation process is simple and straightforward on all major platforms.

The installation is so simple, as a matter of fact, that it can be summed up in three simple steps:

1. Go to the [Node home page](#).
2. Click the big green button that says INSTALL.
3. Follow instructions.

For Windows and OS X, an installer will be downloaded that walks you through the process. For Linux, you will probably be up and running more quickly if you [use a package manager](#).



If you're a Linux user and you do want to use a package manager, make sure you follow the instructions in the aforementioned web page. Many Linux distributions will install an extremely old version of Node if you don't add the appropriate package repository.

You can also [download a standalone installer](#), which can be helpful if you are distributing Node to your organization.

If you have trouble building Node, or for some reason you would like to build Node from scratch, please refer to the [official installation instructions](#).

Using the Terminal

I’m an unrepentant fan of the power and productivity of using a terminal (also called a “console” or “command prompt”). Throughout this book, all examples will assume you’re using a terminal. If you’re not friends with your terminal, I highly recommend you spend some time familiarizing yourself with your terminal of choice. Many of the utilities in this book have corresponding GUI interfaces, so if you’re dead set against using a terminal, you have options, but you will have to find your own way.

If you’re on OS X or Linux, you have a wealth of venerable shells (the terminal command interpreter) to choose from. The most popular by far is bash, though zsh has its adherents. The main reason I gravitate toward bash (other than long familiarity) is ubiquity. Sit down in front of any Unix-based computer, and 99% of the time, the default shell will be bash.

If you’re a Windows user, things aren’t quite so rosy. Microsoft has never been particularly interested in providing a pleasant terminal experience, so you’ll have to do a little more work. Git helpfully includes a “Git bash” shell, which provides a Unix-like terminal experience (it only has a small subset of the normally available Unix command-line utilities, but it’s a useful subset). While Git bash provides you with a minimal bash shell, it’s still using the built-in Windows console application, which leads to an exercise in frustration (even simple functionality like resizing a console window, selecting text, cutting, and pasting is unintuitive and awkward). For this reason, I recommend installing a more sophisticated terminal such as [Console2](#) or [ConEmu](#). For Windows power users—especially for .NET developers or for hardcore Windows systems or network administrators—there is another option: Microsoft’s own PowerShell. PowerShell lives up to its name: people do remarkable things with it, and a skilled PowerShell user could give a Unix command-line guru a run for their money. However, if you move between OS X/Linux and Windows, I still recommend sticking with Git bash for the consistency it provides.

Another option, if you’re a Windows user, is virtualization. With the power and architecture of modern computers, the performance of virtual machines (VMs) is practically indistinguishable from actual machines. I’ve had great luck with Oracle’s free VirtualBox, and Windows 8 offers VM support built in. With cloud-based file storage, such as Dropbox, and the easy bridging of VM storage to host storage, virtualizing is looking more attractive all the time. Instead of using Git bash as a bandage on Windows’s lackluster console support, consider using a Linux VM for development. If you find the

UI isn't as smooth as you would like, you could use a terminal application, such as **PuTTY**, which is what I often do.

Finally, no matter what sytem you're on, there's the excellent **Codio**. Codio is a website that will spin up a new Linux instance for every project you have and provide an IDE and command line, with Node already installed. It's extremely easy to use and is a great way to get started very quickly with Node.



When you specify the `-g` (global) option when installing npm packages, they are installed in a subdirectory of your Windows home directory. I've found that a lot of these packages don't perform well if there are spaces in your username (my username used to be "Ethan Brown," and now it's "ethan.brown"). For your sanity, I recommend choosing a Windows username without a space in it. If you already have such a username, it's advisable to create a new user, and then transfer your files over to the new account: trying to rename your Windows home directory is possible but fraught with danger.

Once you've settled on a shell that makes you happy, I recommend you spend some time getting to know the basics. There are many wonderful tutorials on the Internet, and you'll save yourself a lot of headaches later on by learning a little now. At minimum, you should know how to navigate directories; copy, move, and delete files; and break out of a command-line program (usually `Ctrl-C`). If you want to become a terminal ninja, I encourage you to learn how to search for text in files, search for files and directories, chain commands together (the old "Unix philosophy"), and redirect output.



On many Unix-like systems, `Ctrl-S` has a special meaning: it will "freeze" the terminal (this was once used to pause output quickly scrolling past). Since this is such a common shortcut for Save, it's very easy to unthinkingly press, which leads to a very confusing situation for most people (this happens to me more often than I care to admit). To unfreeze the terminal, simply hit `Ctrl-Q`. So if you're ever confounded by a terminal that seems to have suddenly frozen, try pressing `Ctrl-Q` and see if it releases it.

Editors

Few topics inspire such heated debate among programmers as the choice of editors, and for good reason: the editor is your primary tool. My editor of choice is `vi`¹ (or an editor that has a `vi` mode). `vi` isn't for everyone (my coworkers constantly roll their eyes at me

1. These days, `vi` is essentially synonymous with `vim` (`vi` improved). On most systems, `vi` is aliased to `vim`, but I usually type `vim` to make sure I'm using `vim`.

when I tell them how easy it would be to do what they're doing in vi), but finding a powerful editor and learning to use it will significantly increase your productivity and, dare I say it, enjoyment. One of the reasons I particularly like vi (though hardly the most important reason) is that like bash, it is ubiquitous. If you have access to a Unix system (Cygwin included), vi is there for you. Many popular editors (even Microsoft Visual Studio!) have a vi mode. Once you get used to it, it's hard to imagine using anything else. vi is a hard road at first, but the payoff is worth it.

If, like me, you see the value in being familiar with an editor that's available anywhere, your other option is Emacs. Emacs and I have never quite gotten on (and usually you're either an Emacs person or a vi person), but I absolutely respect the power and flexibility that Emacs provides. If vi's modal editing approach isn't for you, I would encourage you to look into Emacs.

While knowing a console editor (like vi or Emacs) can come in incredibly handy, you may still want a more modern editor. Some of my frontend colleagues swear by Coda, and I trust their opinion. Unfortunately, Coda is available only on OS X. Sublime Text is a modern and powerful editor that also has an excellent vi mode, and it's available on Windows, Linux, and OS X.

On Windows, there are some fine free options out there. TextPad and Notepad++ both have their supporters. They're both capable editors, and you can't beat the price. If you're a Windows user, don't overlook Visual Studio as a JavaScript editor: it's remarkably capable, and has one of the best JavaScript autocomplete engines of any editor. You can download Visual Studio Express from Microsoft for free.

npm

npm is the ubiquitous package manager for Node packages (and is how we'll get and install Express). In the wry tradition of PHP, GNU, WINE, and others, "npm" is not an acronym (which is why it isn't capitalized); rather, it is a recursive abbreviation for "npm is not an acronym."

Broadly speaking, a package manager's two primary responsibilities are installing packages and managing dependencies. npm is a fast, capable, and painless package manager, which I feel is in large part responsible for the rapid growth and diversity of the Node ecosystem.

npm is installed when you install Node, so if you followed the steps listed earlier, you've already got it. So let's get to work!

The primary command you'll be using with npm (unsurprisingly), is `install`. For example, to install Grunt (a popular JavaScript task runner), you would issue the following command (on the console):

```
npm install -g grunt-cli
```

The `-g` flag tells npm to install the package *globally*, meaning it's available globally on the system. This distinction will become clearer when we cover the *package.json* files. For now, the rule of thumb is that JavaScript utilities (like Grunt) will generally be installed globally, whereas packages that are specific to your web app or project will not.



Unlike languages like Python—which underwent a major language change from 2.0 to 3.0, necessitating a way to easily switch between different environments—the Node platform is new enough that it is likely that you should always be running the latest version of Node. However, if you do find yourself needing to support multiple version of Node, there is a project, `nvm`, that allows you to switch environments.

A Simple Web Server with Node

If you've ever built a static HTML website before, or are coming from a PHP or ASP background, you're probably used to the idea of the web server (Apache or IIS, for example) serving your static files so that a browser can view them over the network. For example, if you create the file *about.html*, and put it in the proper directory, you can then navigate to *http://localhost/about.html*. Depending on your web server configuration, you might even be able to omit the *.html*, but the relationship between URL and filename is clear: the web server simply knows where the file is on the computer, and serves it to the browser.



localhost, as the name implies, refers to the computer you're on. This is a common alias for the IPv4 loopback address 127.0.0.1, or the IPv6 loopback address ::1. You will often see 127.0.0.1 used instead, but I will be using *localhost* in this book. If you're using a remote computer (using SSH, for example), keep in mind that browsing to *localhost* will not connect to that computer.

Node offers a different paradigm than that of a traditional web server: the app that you write *is* the web server. Node simply provides the framework for you to build a web server.

“But I don't want to write a web server,” you might be saying! It's a natural response: you want to be writing an app, not a web server. However, Node makes the business of writing

this web server a simple affair (just a few lines, even) and the control you gain over your application in return is more than worth it.

So let's get to it. You've installed Node, you've made friends with the terminal, and now you're ready to go.

Hello World

I've always found it unfortunate that the canonical introductory programming example is the uninspired message "Hello World." However, it seems almost sacrilegious at this point to fly in the face of such ponderous tradition, so we'll start there, and then move on to something more interesting.

In your favorite editor, create a file called *helloWorld.js*:

```
var http = require('http');

http.createServer(function(req,res){
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello world!');
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```

Make sure you are in the same directory as *helloWorld.js*, and type **node helloWorld.js**. Then open up a browser and navigate to *http://localhost:3000*, and voilà! Your first web server. This particular one doesn't serve HTML; rather, it just transmits the message "Hello world!" in plaintext to your browser. If you want, you can experiment with sending HTML instead: just change *text/plain* to *text/html* and change 'Hello world!' to a string containing valid HTML. I didn't demonstrate that, because I try to avoid writing HTML inside JavaScript for reasons that will be discussed in more detail in [Chapter 7](#).

Event-Driven Programming

The core philosophy behind Node is that of *event-driven programming*. What that means for you, the programmer, is that you have to understand what events are available to you and how to respond to them. Many people are introduced to event-driven programming by implementing a user interface: the user clicks on something, and you handle the "click event." It's a good metaphor, because it's understood that the programmer has no control over when, or if, the user is going to click something, so event-driven programming is really quite intuitive. It can be a little harder to make the conceptual leap to responding to events on the server, but the principle is the same.

In the previous code example, the event is implicit: the event that's being handled is an HTTP request. The `http.createServer` method takes a function as an argument; that

function will be invoked every time an HTTP request is made. Our simple program just sets the content type to plaintext and sends the string “Hello world!”

Routing

Routing refers to the mechanism for serving the client the content it has asked for. For web-based client/server applications, the client specifies the desired content in the URL; specifically, the path and querystring (the parts of a URL will be discussed in more detail in [Chapter 6](#)).

Let’s expand our “Hello world!” example to do something more interesting. Let’s serve a really minimal website consisting of a home page, an About page, and a Not Found page. For now, we’ll stick with our previous example and just serve plaintext instead of HTML:

```
var http = require('http');

http.createServer(function(req,res){
  // normalize url by removing querystring, optional
  // trailing slash, and making it lowercase
  var path = req.url.replace(/\/(?:(?:\?.*)?$/ , '').toLowerCase();
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('Homepage');
      break;
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('About');
      break;
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end('Not Found');
      break;
  }
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```

If you run this, you’ll find you can now browse to the home page (<http://localhost:3000>) and the About page (<http://localhost:3000/about>). Any querystrings will be ignored (so <http://localhost:3000/?foo=bar> will serve the home page), and any other URL (<http://localhost:3000/foo>) will serve the Not Found page.

Serving Static Resources

Now that we’ve got some simple routing working, let’s serve some real HTML and a logo image. These are called “static resources” because they don’t change (as opposed to, for example, a stock ticker: every time you reload the page, the stock prices change).



Serving static resources with Node is suitable for development and small projects, but for larger projects, you will probably want to use a proxy server such as Nginx or a CDN to serve static resources. See [Chapter 16](#) for more information.

If you've worked with Apache or IIS, you're probably used to just creating an HTML file, navigating to it, and having it delivered to the browser automatically. Node doesn't work like that: we're going to have to do the work of opening the file, reading it, and then sending its contents along to the browser. So let's create a directory in our project called *public* (why we don't call it *static* will become evident in the next chapter). In that directory, we'll create *home.html*, *about.html*, *404.html*, a subdirectory called *img*, and an image called *img/logo.jpg*. I'll leave that up to you: if you're reading this book, you probably know how to write an HTML file and find an image. In your HTML files, reference the logo thusly: ``.

Now modify *helloWorld.js*:

```
var http = require('http'),
    fs = require('fs');

function serveStaticFile(res, path, contentType, responseCode) {
  if(!responseCode) responseCode = 200;
  fs.readFile(__dirname + path, function(err,data) {
    if(err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end('500 - Internal Error');
    } else {
      res.writeHead(responseCode,
        { 'Content-Type': contentType });
      res.end(data);
    }
  });
}

http.createServer(function(req,res){
  // normalize url by removing querystring, optional
  // trailing slash, and making lowercase
  var path = req.url.replace(/\?(?:\?.*)?$/, '')
    .toLowerCase();
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html');
      break;
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html');
      break;
    case '/img/logo.jpg':
      serveStaticFile(res, '/public/img/logo.jpg',
```

```

        'image/jpeg');
    break;
default:
    serveStaticFile(res, '/public/404.html', 'text/html',
        404);
    break;
}
}).listen(3000);

console.log('Server started on localhost:3000; press Ctrl-C to terminate....');
```



In this example, we're being pretty unimaginative with our routing. If you navigate to `http://localhost:3000/about`, the `public/about.html` file is served. You could change the route to be anything you want, and change the file to be anything you want. For example, if you had a different About page for each day of the week, you could have files `public/about_mon.html`, `public/about_tue.html`, and so on, and provide logic in your routing to serve the appropriate page when the user navigates to `http://localhost:3000/about`.

Note we've created a helper function, `serveStaticFile`, that's doing the bulk of the work. `fs.readFile` is an asynchronous method for reading files. There is a synchronous version of that function, `fs.readFileSync`, but the sooner you start thinking asynchronously, the better. The function is simple: it calls `fs.readFile` to read the contents of the specified file. `fs.readFile` executes the callback function when the file has been read; if the file didn't exist or there were permissions issues reading the file, the `err` variable is set, and the function returns an HTTP status code of 500 indicating a server error. If the file is read successfully, the file is sent to the client with the specified response code and content type. Response codes will be discussed in more detail in [Chapter 6](#).



`__dirname` will resolve to the directory the executing script resides in. So if your script resides in `/home/sites/app.js`, `__dirname` will resolve to `/home/sites`. It's a good idea to use this handy global whenever possible. Failing to do so can cause hard-to-diagnose errors if you run your app from a different directory.

Onward to Express

So far, Node probably doesn't seem that impressive to you. We've basically replicated what Apache or IIS do for you automatically, but now you have some insight into how Node does things and how much control you have. We haven't done anything particularly impressive, but you can see how we could use this as a jumping-off point to do more sophisticated things. If we continued down this road, writing more and more

sophisticated Node applications, you might very well end up with something that resembles Express....

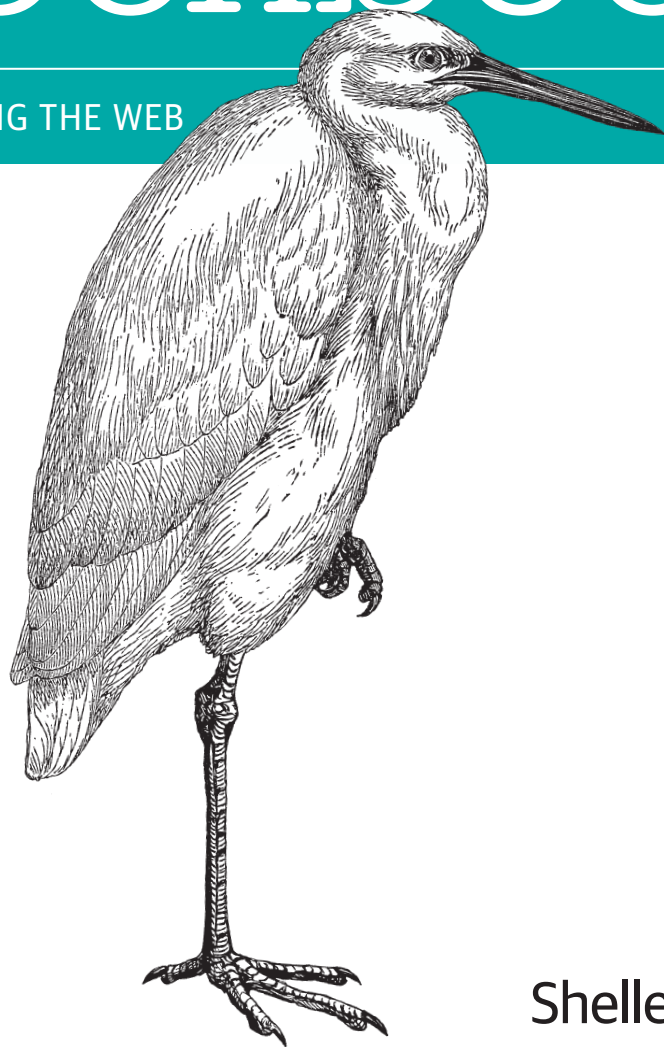
Fortunately, we don't have to: Express already exists, and it saves you from implementing a lot of time-consuming infrastructure. So now that we've gotten a little Node experience under our belt, we're ready to jump into learning Express.

O'REILLY®

2nd Edition

JavaScript Cookbook

PROGRAMMING THE WEB



Shelley Powers

SECOND EDITION

JavaScript Cookbook

Shelley Powers

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Modularizing and Managing JavaScript

One of the great aspects of writing Node.js applications is the built-in modularity the environment provides. As demonstrated in [Chapter 11](#), it's simple to download and install any number of Node modules, and using them is equally simple: just include a single `require()` statement naming the module, and you're off and running.

The ease with which the modules can be incorporated is one of the benefits of JavaScript *modularization*. Modularizing ensures that external functionality is created in such a way that it isn't dependent on other external functionality, a concept known as *loose coupling*. This means I can use a Foo module, without having to include a Bar module, because Foo is tightly dependent on having Bar included.

JavaScript modularization is both a discipline and a contract. The discipline comes in by having to follow certain mandated criteria in order for external code to participate in the module system. The contract is between you, me, and other JavaScript developers: we're following an agreed on path when we produce (or consume) external functionality in a module system, and we all have expectations based on the module system.



ECMAScript 6 provides native support for modules, but the specification is still undergoing change and there is no implementation support yet. There is some [support for it in Traceur](#), as well as a [polyfill](#), which can at least provide an idea of how they'll be implemented in the future.

Chances are you have used modularized JavaScript. If you have used jQuery with RequireJS or Dojo, you've used modularized JavaScript. If you've used Node, you've used a modular system. They don't look the same, but they work the same: ensuring that functionality developed by disparate parties works together seamlessly. The modular system that RequireJS and Dojo support is the Asynchronous Module Definition

(AMD), while Node's system is based on CommonJS. One major difference between the two is that AMD is asynchronous, while CommonJS is synchronous.

Even if you don't use a formal modular system, you can still improve the performance of script loading with script loaders and using new HTML5 `async` functionality. You can also improve the management of your entire application process using tools such as Grunt, or ensuring your own code is packaged for ease of use and innovation.



One major dependency on virtually all aspects of application and library management and publication is the use of Git, a source control system, and GitHub, an extremely popular Git *endpoint*. How Git works and using Git with GitHub are beyond the scope of this book. I recommend *The Git Pocket Guide* (O'Reilly) to get more familiar with Git, and GitHub's [own documentation](#) for more on using this service.

12.1. Loading Scripts with a Script Loader

Problem

You need to use several different JavaScript libraries in your web pages, and they're starting to slow the page loads.

Solution

One solution is to use a *script loader* to load your JavaScript files asynchronously and concurrently. Examples of use are documented in the discussion.

Discussion

There are several techniques you can use to load JavaScript files. One is the traditional method of using a script element for each file, and just loading each in turn. The issue that people have had with this approach is the inefficiency of having to access each file individually, the problems that can occur if scripts are loaded out of order (with one script being dependent on another already loaded), and the fact that the entire page is blocked while the scripts load.

Some solutions are to compile all the individual JavaScript files into a single file, which is what the content management system (CMS) Drupal does. This eliminates the multiple file access and even the issues with ordering, but it still leaves us with the fact that the page is blocked from loading until the scripts are loaded.

Script loaders were created to provide a way of loading JavaScript files asynchronously, which means the rest of the page can continue loading while the script is loading. They

use *script injection*: creating a script element in a script block that loads the JavaScript file, and then appending that block to the page. The *inline* JavaScript is executed asynchronously and does not block the page from loading like the use of the traditional script element does.

The code to do so can be similar to the script block shown in the following minimal HTML5 page:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>title</title>
</head>
<body>
  <script>
    var script = document.querySelector("script");
    var t = document.createElement("script");
    t.src = "test1.js";
    script.parentNode.insertBefore(t,script);
  </script>
</body>
</html>
```

To prevent the variables from cluttering up the global namespace, they can be included in an Immediately-Invoked Function Expression (IIFE):

```
<script>
(function() {
  var script = document.querySelector("script");
  var t = document.createElement("script");
  t.src = "test1.js";
  script.parentNode.insertBefore(t,script);
})();
</script>
```

If you need to use a pathname for the script, you can use a protocol-relative URL (sometimes referred to as a *protocol-less URL*) so that the code adapts whether the page is accessed with *http* or *https*:

```
t.src = "//somecompany.com/scriptfolder/test1.js";
```

With this, the client application uses the same protocol (*http* or *https*) used to access the parent page.

Multiple scripts can be loaded into the page using this approach. It can also be used to load CSS files, as well as larger images or other media files. However, we don't have to do the work ourselves: we can use a script loading library, such as HeadJS.

According to the HeadJS documentation, the best approach to including support for the library is to include a link to the library in the head element:

```

<html>
  <head>
    <script src="head.min.js"></script>
    <script>
      head.load("file1.js", "file2.js");
    </script>
  </head>
  <body>
    <!-- my content-->

    <script>
      head.ready(function () {
        // some callback stuff
      });
    </script>
  </body>
</html>

```

Note the `head.load()` function call. All of the script files to be loaded are listed in the function call. In addition, any ready state functionality can be provided in the `head.ready()` function call.

If you do have JavaScript, you want to load right away; rather than using another script element, you can use a *data-* attribute on the script element loading HeadJS:

```
<script src="head.min.js" data-headjs-load="init.js"></script>
```

Any immediately invoked functionality is then listed in *init.js*.



HeadJS has other functionality, including assistance for responsive design and browser version support. Read more about setting it up in the [set up documentation](#).

Another script loader with an interesting twist is *Basket.js*. It also loads JavaScript files asynchronously, but it goes a step further: it caches the script using `localStorage`, which means if the JavaScript has already been accessed once, a second access loads the JavaScript from cache rather than loading the file again.

Once you include the *Basket.js* JavaScript file, you can then define the JavaScript files to be loaded:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>title</title>
  </head>
  <body>
    <script src="basket.full.min.js"></script>

```

```

<script>
  basket.require({ url: 'test1.js'},
                { url: 'test2.js'});
</script>
</body>
</html>

```

If you monitor the page using your browser's debugger/development tools, and reload the page, you'll note that the files aren't accessed again after the first load.

To handle source dependencies, Basket.js returns a *promise* from `require()`, and the `then()` callback is executed. You can then list the second JavaScript file in the callback:

```

<script>
  basket.require({ url: 'test2.js'}).then(function() {
    basket.require({ url: 'test1.js'});
  });
</script>

```



Access Basket.js and read how to use it in the library's [home page](#).

12.2. Loading Scripts Asynchronously the HTML5 Way

Problem

You're interested in processing scripts asynchronously—not blocking the page from loading while the scripts load—but you have discovered that the *script injection* technique has one problem: the CSS Object Model (CSSOM) blocks inline scripts because these scripts typically operate on the CSSOM. Since the CSSOM doesn't know what the script is going to do, it blocks the script until all of the CSS is loaded. This, then, delays the network access of the script until all CSS files have been loaded.

Solution

Use the new HTML5 `async` script element attribute instead of script injection:

```

<script src="//cdnjs.cloudflare.com/ajax/libs/mathjs/0.26.0/math.min.js" async>
</script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/backbone-min.js" async>
</script>

```

Discussion

There are two script element attributes: `defer`, which defers script loading until the rest of the page is loaded, and the newest `async`. The latter tells the browser to load the script asynchronously, as the page is being parsed. It only works with external scripts; the page still blocks with inline scripts.

The `async` attribute prevents many of the problems we've had with blocked scripts and having to use tricks such as script injection. The only reason script injection is still being used is there are older versions of browsers, such as IE9 and older, that don't support it.

12.3. Converting Your JavaScript to AMD and RequireJS

Problem

You're interested in taking advantage of modularization and controlled dependencies by converting your libraries to the Asynchronous Module Definition (AMD) format, implemented with RequireJS, but you're not sure where to start and what to do.

Solution

RequireJS is integrated into the following three small JavaScript libraries:

one.js

```
define(function() {  
    return {  
        hi: function() {  
            console.log('hello from one');  
        }  
    }  
});
```

two.js

```
define(function() {  
    return {  
        hi: function(val) {  
            console.log('hello ' + val + ' from two');  
        }  
    }  
});
```

mylib.js

```
require(["./one", "./two"], function(one, two) {  
    one.hi();  
    two.hi('world');  
    console.log("And that's all");  
});
```

And the web page, *index.html*:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Modularization</title>
    <script data-main="scripts/mylib" src="scripts/require.js"></script>
  </head>
  <body>
    <h1>Stuff</h1>
  </body>
</html>
```

Discussion

Consider the following three very basic JavaScript libraries:

one.js

```
function oneHi() {
  console.log('hello from one');
}
```

two.js

```
function twoHi(val) {
  console.log('hello ' + val + ' from two');
}
```

mylib.js

```
function allThat() {
  oneHi();
  twoHi('world');
  console.log("And that's all");
}
```

They could be included in a simple web page as demonstrated in the following code, assuming all the JavaScript libraries are in a subdirectory named *scripts/*:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Modularization</title>
    <script src="scripts/one.js" type="text/javascript"></script>
    <script src="scripts/two.js" type="text/javascript"></script>
    <script src="scripts/mylib.js" type="text/javascript"></script>
    <script type="text/javascript">
      allThat();
    </script>
  </head>
  <body>
    <h1>Stuff</h1>
```



```
</body>
</html>
```

And you might expect the application to work, with the messages printed out in the right order. However, if you make a modest change, such as use the `async` attribute with all of the scripts:

```
<script src="scripts/one.js" async type="text/javascript"></script>
<script src="scripts/two.js" async type="text/javascript"></script>
<script src="scripts/mylib.js" async type="text/javascript"></script>
```

You'll be hosed, because the browser no longer blocks program execution, waiting for each script to load, in turn, before going to the next. Other challenges that can occur are that you're using other people's libraries and you don't know the correct order to list the source scripts, or you forget one or more of them. The problem with this common approach from the past is that nothing enforces both order and dependencies. That's where `RequireJS` comes in.

In the solution, you'll notice two key words: `define` and `require`. The `define` keyword is used to define a module, while `require` is used to list dependencies with a callback function that's called when all dependencies are loaded.

In the solution, two of the libraries are defined as modules, each return a function. The third library, *mylib.js*, declares the two modules as dependencies and in the callback function, invokes the returned module functions. All of this is pulled into the HTML page with the following line:

```
<script data-main="scripts/mylib" src="scripts/require.js"></script>
```

The actual source is the `RequireJS` library. The custom attribute `data-main` specifies the JavaScript source to load after `RequireJS` is loaded.

The modules can return more than one function, or can return data objects, functions, or a combination of both:

```
define(function() {
  return {
    value1: 'one',
    value2: 'two',
    doSomething: function() {
      // do something
    }
  }
})
```

Modules can also have dependencies. The following code version of *two.js* creates a dependency on *one.js* in *two.js* and removes it as a dependency in *mylib.js*:

two.js

```
define(['one'], function(one) {  
    return {  
        hi: function(val) {  
            one.hi();  
            console.log('hello ' + val + ' from two');  
        }  
    };  
});
```

mylib.js

```
require(["./two"],function(two) {  
    two.hi('world');  
    console.log("And that's all");  
});
```



Typically after you create your JavaScript files, you'll want to optimize them. RequireJS provides the tools and documentation for optimizing your source at <http://requirejs.org/docs/optimization.html>.

See Also

Your library can still exist as a standard JavaScript library and an AMD-compliant module, as discussed in [Recipe 12.9](#).

12.4. Using RequireJS with jQuery or Another Library

Problem

Your applications uses jQuery (or Underscore.js or Backbone). How can the library fit into the use of RequireJS to manage dependencies?

Solution

If the library can work with AMD (as jQuery can), and you save the jQuery file as *jquery.js* and load it in the same directory as your application JavaScript, you can use the jQuery functionality easily, as shown in the following small code snippet:

```
require(["./jquery"],function($) {  
    $('h1').css('color','red');  
});
```

However, if the jQuery file is named something else, or you're accessing the library from a CDN, then you'll need to use a RequireJS *shim*:

```
requirejs.config({
  baseUrl: 'scripts/lib',
  paths: {
    jquery: '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min'
  },
});
```

Discussion

As the solution demonstrates, if your application code already incorporates jQuery's dollar sign (\$) and the jQuery file is local to the script, you can incorporate its use in your application in the same manner used for any other module. The jQuery library can recognize that it's within a RequireJS environment, and respond accordingly. Where things get a little more complicated is if the library is not accessed locally, is accessed from a CDN, or the library doesn't support AMD.

To demonstrate, I modified the source files discussed in [Recipe 12.3](#). The source files are now organized in the following directory structure:

```
www
  app
    main.js
  index.html
  scripts
    app.js
  lib
    one.js
    require.js
    two.js
```

In addition, I removed the `define()` in the source library `two.js`, making it into an *anonymous closure*—an IIFE object that is added to the Window object as two:

```
(function (){
  window.two = this;
  this.hi = function(val) {
    console.log('hello ' + val + ' from two');
  }
})();
```

The `one.js` file still contains the AMD `define()` statement, meaning it requires no special handling to use:

```
define(function() {
  return {
    hi: function() {
      console.log('hello from one');
    }
  }
});
```

The *app.js* file contains a RequireJS config block that, among other things, sets a base URL for all loaded modules, defines a CDN path for both jQuery and the app subdirectory, and creates a shim for the non-AMD compliant two. It also loads the *app/main* module:

```
requirejs.config({
  baseUrl: 'scripts/lib',
  paths: {
    app: '../..app',
    jquery: '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min'
  },
  shim: {
    two: {
      exports: 'two'
    }
  }
});

requirejs(["app/main"]);
```

The shim for two defines an exported object (an object defined on Window in the browser), since the library doesn't use `define()` to identify the object.

Lastly, the *main.js* module lays out the dependency on jQuery, one, and two, and runs the application:

```
define(["jquery", "one", "two"], function($, one, two) {
  one.hi();
  two.hi('world');
  console.log("And that's all");
  $('h1').css('color', 'red');
});
```

If two had been dependent on one of the modules or other libraries, such as one, the dependency would have been noted in the shim:

```
requirejs.config({
  baseUrl: 'scripts/lib',
  paths: {
    app: '../..app',
    jquery: '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min'
  },
  shim: {
    two: {
      deps: ['one'],
      exports: 'two'
    }
  }
});
```

If you'd like to make your JavaScript library into an AMD-compliant module, but still allow it to be used in other contexts, you can add a small amount of code to ensure both:

```

(function (){
  window.two = this;
  this.hi = function(val) {
    console.log('hello ' + val + ' from two');
  }
})();

```

The tiny library is now redesigned into an IIFE. Any private data and methods would be fully enclosed in the closure, and the only public method is exposed by adding it as a property to the object. The object itself is given global access via assignment to the `Window` property.

A variation on this would be the following, where the exposed methods and data are returned as an object to the assigned variable:

```

var two = (function (){
  return {
    hi: function (val) {
      console.log('hello ' + val + ' from two');
    }
  }
})();

```

The code now meets the *module pattern*, ensuring both public and private data and functions are encapsulated using the closure, and globally accessible methods and data are returned in the object. Another variation of the module pattern is the following:

```

var two = (function() {
  var my = {};
  my.hi = function(val) {
    console.log('hello ' + val + ' from two');
  };
  return my;
})();

```

I modified the original form of the object to make it AMD compliant:

```

(function (){
  window.two = this;
  this.hi = function(val) {
    console.log('hello ' + val + ' from two');
  }

  if ( typeof define === "function" && define.amd ) {
    define( "two", [], function() {
      return two;
    });
  }
})();

```

The code tests to see if the `define()` function exists. If so, then it's invoked, passing in the name of the exported library object and in the callback, returning the exported library object. This is how a library such as jQuery can work in AMD, but still work in other traditional JavaScript environments.

A variation, using the more established module pattern, is the following:

```
var two = (function (){
    var two = {};

    two.hi = function(val) {
        console.log('hello ' + val + ' from two');
    }

    if ( typeof define === "function" && define.amd ) {
        define( "two", [], function() {
            return two;
        });
    }

    return two;
})();
```



jQuery also supports the CommonJS modular system.

12.5. Loading and Using Dojo Modules

Problem

You're interested in using some of the Dojo functionality, but you're not sure how to load the associated modules.

Solution

Dojo has implemented the AMD architecture for its functionality. When you add the main Dojo script to your page, what you're loading is the module loader, rather than all of its various functions:

```
<script src="http://ajax.googleapis.com/ajax/libs/dojo/1.10.0/dojo/dojo.js"
    data-dojo-config="async: true"></script>
```

The library can be accessed at a CDN, as the code snippet demonstrates. The custom data attribute `data-dojo-config` specifies that the Dojo asynchronous AMD loader should be used.

To use the Dojo functionality, specify the dependencies in the `require()` method:

```
<script>
  require([
    'dojo/dom',
    'dojo/dom-construct'
  ], function (dom, domConstruct) {
    var ph = dom.byId("placeholder");
    ph.innerHTML = "Using Dojo";
    domConstruct.create("h1", {innerHTML: "<i>Howdy!</i>"}, ph, "before");
  });
</script>
```

Discussion

Dojo is a sophisticated library system providing functionality similar to that provided in the jQuery environment. It does require a little time to become familiar with its implementation of AMD, though, before jumping in.

In the solution, the Dojo asynchronous loader is sourced from a CDN. The solution then imports two Dojo modules: `dojo/dom` and `dojo/dom-construct`. Both provide much of the basic DOM functionality, such as the ability to access an existing element by an identifier (`dom.byId()`), and create and place a new element (`domConstruct.create()`). To give you a better idea how it all holds together, a complete page example is given in [Example 12-1](#).

Example 12-1. A complete Dojo example accessing one page element and adding another

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Dojo</title>
  <script src="http://ajax.googleapis.com/ajax/libs/dojo/1.10.0/dojo/dojo.js"
    data-dojo-config="async: true"></script>
</head>
<body>
  <div id="placeholder"></div>
  <script>
    require([
      'dojo/dom',
      'dojo/dom-construct'
    ], function (dom, domConstruct) {
      var ph = dom.byId("placeholder");
      ph.innerHTML = "Using Dojo";
      domConstruct.create("h1", {innerHTML: "<i>Howdy!</i>"}, ph, "before");
    });
  </script>
</body>
</html>
```



Though Dojo is generally AMD-compatible, there's still some funkiness with the implementation that makes it incompatible with a module loader like RequireJS. The *concepts* of a module loader, the `require()` and `define()` functions, and creating a configuration object are the same, but implementation compatibility fails.

Dojo does provide a **decent set of tutorials** to help you understand more fully how the framework operates.

12.6. Installing and Maintaining Node Modules with npm

Problem

You're new to Node. You've installed it, and played around with the core Node modules installed with Node. But now, you need something more.

Solution

The glue that holds the Node universe together is npm, the Node package manager. To install a specific module, use the following on the command line:

```
npm install packagename
```

If you want to install the package globally, so it's accessible from all locations in the computer, use the following:

```
npm install -g packagename
```

When to install locally or globally is dependent on whether you're going to `require()` the module, or if you need to run it from the command line. Typically you install `require()` modules locally, and executables are installed globally, though you don't *have* to follow this typical usage. If you do install a module globally, you might need administrative privileges:

```
sudo npm install -g packagename
```

Discussion

The solution demonstrated the most common use of npm: installing a registered npm module locally or globally on your system. However, you can install modules that are located in GitHub, downloaded as a tar file, or located in a folder. If you type:

```
npm install --help
```

you'll get a list of allowable approaches for installing a module:

```
npm install
npm install <pkg>
npm install <pkg>@<tag>
```



```
npm install <pkg>@<version>
npm install <pkg>@<version range>
npm install <folder>
npm install <tarball file>
npm install <tarball url>
npm install <git:// url>
npm install <github username>/<github project>
```

If your current directory contains a *npm-shrinkwrap.json* or *package.json* file, the dependencies in the files are installed by typing `npm install`.



Recipe 12.10 covers the structure and purpose of the *package.json* file.

To remove an installed Node module, use:

```
npm rm packagename
```

The package and any dependencies are removed. To update existing packages, use:

```
npm update [g] [packagename [packagename ...]]
```

You can update locally or globally installed modules. When updating, you can list all modules to be updated, or just type the command to update all locally-installed modules relative to your current location.

12.7. Searching for a Specific Node Module via npm

Problem

You're creating a Node application and want to reuse existing modules, but you don't know how to discover them.

Solution

In most cases, you'll discover modules via recommendations from your friends and co-developers, but sometimes you need something new.

You can search for new modules directly at the [npm website](#). The front page also lists the most popular modules, which are worth an exploratory look.

You can also use npm directly to search for a module. For instance, if you're interested in modules that do something with PDFs, run the following search at the command line:

```
npm search pdf
```

Discussion

The [npm website](#) provides more than just good documentation for using npm; it also provides a listing of newly updated modules, as well as those modules most depended on. Regardless of what you're looking for, you definitely should spend time exploring these essential modules. In addition, if you access each module's page at npm, you can see how popular the module is, what other modules are dependent on it, the license, and other relevant information.

However, you can also search for modules, directly, using npm.

The first time you perform a search with npm, you'll get the following feedback:

```
npm WARN Building the local index for the first time, please be patient
```

The process can take a fair amount of time, too. Luckily, the index build only needs to be performed the first time you do a search. And when it finishes, you're likely to get a huge number of modules in return, especially with a broader topic such as modules that work with PDFs.

You can refine the results by listing multiple terms:

```
npm search PDF generation
```

This query returns a much smaller list of modules, specific to PDF generation. You can also use a regular expression to search:

```
npm search \/Firefox\\sOS
```

Now I'm getting all modules that reference `Firefox OS`. However, as the example demonstrates, you have to incorporate escape characters specific to your environment, as I did with the beginning of the regular expression, and the use of `\s` for white space.

Once you do find a module that sounds interesting, you can get detailed information about it with:

```
npm view node-firefoxos-cli
```

You'll get the *package.json* file for the module, which can tell you what it's dependent on, who wrote it, and when it was created. I still recommend checking out the module's GitHub page directly. There you'll be able to determine if the module is being actively maintained or not. If you access the npm website page for the module, you'll also get an idea of how popular the module is.

12.8. Converting Your Library into a Node Module

Problem

You want to use one of your libraries in Node.

Solution

Convert the library into a Node module. For example, if the library is designed as the following IIFE:

```
(function () {  
    var val = 'world';  
    console.log('Hello ' + val + ' from two');  
})();
```

You can convert it to work with Node by the simple addition of an exports keyword:

```
module.exports = (function () {  
    return {  
        hi: function(val) {  
            console.log('Hello ' + val + ' from two');  
        }  
    };  
})();
```

You can then use the module in your application:

```
var two = require('./two.js');  
  
two.hi('world');
```

Discussion

Node's module system is based on CommonJS, the second modular system covered in this chapter. CommonJS uses three constructs: exports to define what's exported from the library, require() to include the module in the application, and module, which includes information about the module but also can be used to export a function, directly.

Though the solution maintains the IIFE, it's not really required in the CommonJS environment, because every module operates in its own module space. The following is also acceptable:

```
module.exports.hi = function (val) {  
    console.log('hello ' + val + ' from two');  
}
```

If your library returns an object with several functions and data objects, you can assign each to the comparably named property on module.exports, or you could return an object from a function:

```
module.exports = function () {  
    return {  
        somedata: 'some data',  
        hi: function(val) {  
            console.log('Hello ' + val + ' from two');  
        }  
    }  
}
```

```
    };  
  };
```

And then invoke the object in the application:

```
var twoObj = require('./two.js');  
  
var two = twoObj();  
two.hi(two.somedata);
```

Or you can access the object property directly:

```
var hi = require('./twob.js').hi;  
  
hi('world');
```

Because the module isn't installed using npm, and just resides in the directory where the application resides, it's accessed by the location and name, not just the name.

See Also

In [Recipe 12.9](#), I cover how to make sure your library code works in all of the environments: CommonJS, Node, AMD, and as a traditional JavaScript library.

12.9. Taking Your Code Across All Module Environments

Problem

You've written a library that you'd like to share with others, but folks are using a variety of module systems to incorporate external JavaScript. How can you ensure your library works in all of the various environments?

Solution

The following library with two functions:

```
function concatArray(str, array) {  
  return array.map(function(element) {  
    return str + ' ' + element;  
  });  
}  
  
function splitArray(str,array) {  
  return array.map(function(element) {  
    var len = str.length + 1;  
    return element.substring(len);  
  });  
}
```

Will work with RequireJS, Node, as a plain script, and CommonJS in the browser when converted to:

```
(function(global) {
  'use strict';

  var bbArray = {};

  bbArray.concatArray = function (str, array) {
    return array.map(function(element) {
      return str + ' ' + element;
    });
  };

  bbArray.splitArray = function (str,array) {
    return array.map(function(element) {
      var len = str.length + 1;
      return element.substring(len);
    });
  };

  if (typeof module != 'undefined' && module.exports) {
    module.exports = bbArray;
  } else if ( typeof define === "function" && define.amd ) {
    define( "bbArray", [], function() {
      return bbArray;
    });
  } else {
    global.bbArray = bbArray;
  }
})(this));
```

Discussion

To ensure your library works in a traditional scripting environment, you should encapsulate your functionality in an IIFE, to minimize leak between private and public functionality and data. You'll also want to limit pollution of the global space:

```
(function(global) {
  'use strict';

  var bbArray = {};

  bbArray.concatArray = function (str, array) {
    return array.map(function(element) {
      return str + ' ' + element;
    });
  };

  bbArray.splitArray = function (str,array) {
    return array.map(function(element) {
```

```

        var len = str.length + 1;
        return element.substring(len);
    });

    global.bbArray = bbArray;

})(this));

```

The object is being used in an environment that may not have access to a window object, so the global object (global in Node, window in the browser) is passed as an argument to the object as this, and then defined as global in the library.

At this point, the library can work as a traditional library in a browser application:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Array test</title>
  <script src="bbarray.js" type="text/javascript">
  </script>
  <script type="text/javascript">
    var a = ['one', 'two', 'three'];
    var b = bbArray.concatArray('number is ',a);
    console.log(b);
    var c = bbArray.splitArray('number is ', b);
    console.log(c);
  </script>
</head>
<body>
</body>
</html>

```

The result is two print outs to the console:

```

[ 'number is one', 'number is two', 'number is three' ]
[ 'one', 'two', 'three' ]

```

Next, we'll add the Node support. We add this using the following lines of code:

```

if (typeof module !== 'undefined' && module.exports) {
  module.exports = bbArray;
}

```

This code checks whether the module object is defined and if it is, whether the module.exports object exists. If the tests succeed, then the object is assigned to module.exports, no different than defining exported functionality (covered earlier in [Recipe 12.8](#)). It can now be accessed in a Node application like the following:

```

var bbArray = require('./bbarray.js');

var a = ['one', 'two', 'three'];

```

```

var b = bbArray.concatArray('number is ',a);
console.log(b);
var c = bbArray.splitArray('number is ', b);
console.log(c);

```

Now we add support for CommonJS, specifically RequireJS. From [Recipe 12.4](#), we know to check if `define` exists, and if so, to add support for RequireJS. After adding this modification, the library module now looks like this:

```

(function(global) {
    'use strict';

    var bbArray = {};

    bbArray.concatArray = function (str, array) {
        return array.map(function(element) {
            return str + ' ' + element;
        });
    };

    bbArray.splitArray = function (str,array) {
        return array.map(function(element) {
            var len = str.length + 1;
            return element.substring(len);
        });
    };

    if (typeof module !== 'undefined' && module.exports) {
        module.exports = bbArray;
    } else if ( typeof define === "function" && define.amd ) {
        define( "bbArray", [], function() {
            return bbArray;
        });
    } else {
        global.bbArray = bbArray;
    }

})(this));

```

The module can now be used in a web application that incorporates RequireJS for module support. Following RequireJS's suggestion that all inline scripts be pulled into a separate file, the JavaScript application to test the library is created in a file named *main.js*:

```

require(["./bbarray"], function(bbArray) {
    var a = ['one', 'two', 'three'];
    var b = bbArray.concatArray('number is ',a);
    console.log(b);
    var c = bbArray.splitArray('number is ', b);
    console.log(c);
});

```

And the web page incorporates the RequireJS script, loaded via CDN:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Array test</title>
  <script src="//cdnjs.cloudflare.com/ajax/libs/require.js/2.1.14/require.min.js"
    data-main="main">
  </script>
</head>
<body>

</body>
</html>
```

Modify the URL for Require.js to match what's available at the CDN when you run the test.

See Also

The example covered in this recipe works in all of our environments but it has one limitation: it's not using any other libraries. So what happens when you need to include libraries?

This is where things can get ugly. We know that CommonJS/Node import dependencies with `require`:

```
var library = require('somelib');
```

While AMD incorporates dependencies in `require` or `define`:

```
define(['./somelib'], function(library) {

  // rest of the code
});
```

Not compatible. At all. The workaround for this problem has been either to use Browserify (covered in [Recipe 12.12](#)) or to incorporate a *Universal Module Definition* (UMD). You can see examples of a UMD online, and it's covered in detail in Addy Osmani's "[Writing Modular JavaScript with AMD, CommonJS, and ES Harmony](#)".

12.10. Creating an Installable Node Module

Problem

You've either created a Node module from scratch, or converted an existing library to one that will work in the browser or in Node. Now, you want to know how to modify it into a module that can be installed using npm.

Solution

Once you've created your Node module and any supporting functionality (including module tests), you can package the entire directory. The key to packaging and publishing the Node module is creating a *package.json* file that describes the module, any dependencies, the directory structure, what to ignore, and so on.

The following is a relatively basic *package.json* file:

```
{
  "name": "bbArray",
  "version": "0.1.0",
  "description": "A description of what my module is about",
  "main": "./lib/bbArray",
  "author": {
    "name": "Shelley Powers"
  },
  "keywords": [
    "array",
    "utility"
  ],
  "repository": {
    "type": "git",
    "url": "https://github.com/accountname/bbarray.git"
  },
  "engines" : {
    "node" : ">=0.10.3 <0.12"
  },
  "bugs": {
    "url": "https://github.com/accountname/bbarray/issues"
  },
  "licenses": [
    {
      "type": "MIT",
      "url": "https://github.com/accountname/bbarray/raw/master/LICENSE"
    }
  ],
  "dependencies": {
    "some-module": "~0.1.0"
  },
  "directories":{
    "doc": "./doc",
    "man": "./man",
    "lib": "./lib",
    "bin": "./bin"
  },
  "scripts": {
    "test": "nodeunit test/test-bbarray.js"
  }
}
```

Once you've created *package.json*, package all the source directories and the *package.json* file as a gzipped tarball. Then install the package locally, or install it in npm for public access.

Discussion

The *package.json* file is key to packaging a Node module up for local installation or uploading to npm for management. At a minimum, it requires a **name** and a **version**. The other fields given in the solution are:

- **description**: A description of what the module is and does
- **main**: Entry module for application
- **author**: Author(s) of the module
- **keywords**: List of keywords appropriate for module
- **repository**: Place where code lives, typically GitHub
- **engines**: Node version you know your module works with
- **bugs**: Where to file bugs
- **licenses**: License for your module
- **dependencies**: Any module dependencies
- **directories**: A hash describing directory structure for your module
- **scripts**: A hash of object commands that are run during module lifecycle

There are a host of other options, which are described at the [npm website](#). You can also use a tool to help you fill in many of these fields. Typing the following at the command line runs the tool that asks questions and then generates a basic *package.json* file:

```
npm init
```

Once you have your source set up and your *package.json* file, you can test whether everything works by running the following command in the top-level directory of your module:

```
npm install . -g
```

If you have no errors, then you can package the file as a gzipped tarball. At this point, if you want to publish the module, you'll first need to add yourself as a user in the npm registry:

```
npm add-user
```

To publish the Node module to the npm registry, use the following in the root directory of the module, specifying a URL to the tarball, a filename for the tarball, or a path:

```
npm publish ./
```

If you have development dependencies for your module, such as using a testing framework like Mocha, one excellent shortcut to ensure these are added to your *package.json* file is to use the following, in the same directory as the *package.json* file, when you're installing the dependent module:

```
npm install -g mocha --save-dev
```

Not only does this install Mocha (discussed later, in [Recipe 12.13](#)), this command also updates your *package.json* file with the following:

```
"devDependencies": {  
  "grunt": "^0.4.5",  
  "grunt-contrib-jshint": "^0.10.0",  
  "mocha": "^1.21.4"  
}
```

You can also use this same type of option to add a module to dependencies in *package.json*. The following:

```
npm install d3 --save
```

adds the following to the *package.json* file:

```
"dependencies": {  
  "d3": "^3.4.11"  
}
```

If the module is no longer needed and shouldn't be listed in *package.json*, remove it from the devDependencies with:

```
npm remove mocha --save-dev
```

And remove a module from dependencies with:

```
npm remove d3 --save
```

If the module is the last in either dependencies or devDependencies, the property isn't removed. It's just set to an empty value:

```
"dependencies": {}
```



npm provides a [decent developer guide for creating and installing a Node module](#). You should consider the use of an *.npmignore* file for keeping stuff *out* of your module. And though this is beyond the scope of the book, you should also become familiar with Git and GitHub, and make use of it for your applications/modules.

Extra: The README File and Markdown Syntax

When you package your module or library for reuse and upload it to a source repository such as GitHub, you'll need to provide how-to information about installing the module/library and basic information about how to use it. For this, you need a README file.

You’ve seen files named *README.md* or *readme.md* with applications and Node modules. They’re text-based with some odd, unobtrusive markup that you’re not sure is useful, until you see it in a site like GitHub, where the README file provides all of the project page installation and usage information. The markup translates into HTML, making for readable Web-based help.

The content for the README is marked up with annotation known as Markdown. The popular website Daring Fireball calls Markdown easy to read and write, but “Readability, however, is emphasized above all else.” Unlike with HTML, the Markdown markup doesn’t get in the way of reading the text.



Daring Fireball also provides an [overview of generic Markdown](#), but if you’re working with GitHub files, you might also want to check out [GitHub’s Flavored Markdown](#).

In [Recipe 18.5](#) in [Chapter 18](#), I created a simple Firefox OS mobile app named “Where Am I?” Part of its installation is a *README.md* file that provides information about using the app. The following is a brief excerpt from the file:

```
# Where Am I?
```

```
This is a simple demonstration Firefox OS app that uses the Geolocation API  
to get the user's current location, and then loads a static map into the page.
```

```
## Obtaining
```

```
The Where Am I? app is hosted on the web, in a [Burningbird work directory]  
(http://burningbird.net/work/whereami)
```

```
## Usage
```

```
Import it into the Mozilla WebIDE using the hosted app option, and then run  
the app in one or more simulators.
```

When I use a CLI tool like **Pandoc**, I can convert the *README.md* file into readable HTML:

```
pandoc README.md -o readme.html
```

[Figure 12-1](#) displays the generated content. It’s not fancy, but it is imminently readable.

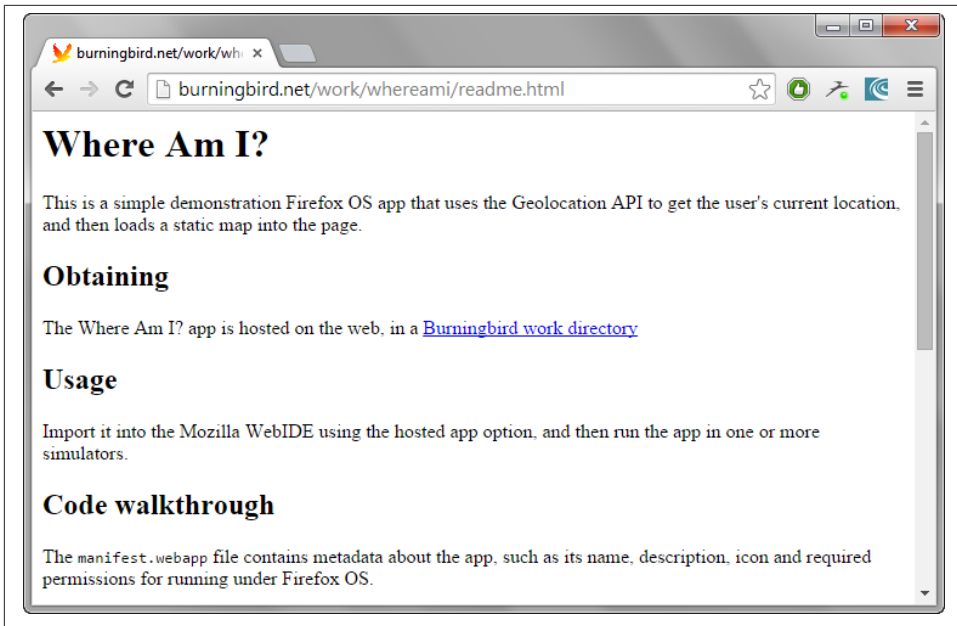


Figure 12-1. Generated HTML from README.md text and Markdown annotation

When you install your source in a site such as GitHub (discussed in [Recipe 7.12](#) in [Chapter 7](#)), GitHub uses the *README.md* file to generate the cover page for the repository.

12.11. Packaging and Managing Your Client-Side Dependencies with Bower

Problem

You really like how npm manages dependencies and wish there was something comparable for the client.

Solution

Bower can help you manage client dependencies. To use it you must have Node, npm, and support for Git installed on your client or server.

Once your environment is set up, install Bower using npm:

```
npm install -g bower
```

Now, to add packages to the *bower-components* subdirectory, install them with bower:

```
bower install jquery
```

Then you can create a *bower.json* file by typing the following in the root directory of your library or application:

```
bower init
```

The application asks a set of questions and generates a *bower.json* file, which can be used to install the dependencies with another simple command:

```
bower install
```

Discussion

Bower is a way of keeping your script and other dependencies collected and up to date. Unlike npm, it can work with a variety of file extensions, including CSS, images, as well as script. You can use it to install dependencies in *bower-components*, and then access the dependencies directly in your web applications:

```
<script src="path/to/bower_components/d3/d3.min.js"></script>
```

You can package all of your application's dependencies in a *bower.json* file, and reinstall them in a fresh directory with a simple command (in the same directory as the *bower.json* file):

```
bower install
```

To ensure you're using the latest and greatest version of the module and library, update your dependencies:

```
bower update
```

If your application is publicly available on GitHub, you can register its dependencies in Bower by, first, ensuring the *bower.json* file for the application is accurate, you're using **semantic versioning** with your Git tags, your application is publicly available as a Git end point (such as GitHub), and the package name adheres to the *bower.json* specification. Once these dependencies are met, register the application:

```
bower register <package-name> <git-endpoint>
```

If you're wondering why you can't use something like `require` directly with Bower, remember that it's a dependency management tool, just like npm. It's the libraries and infrastructure in place, such as RequireJS, that allows you to use modular AMD or CommonJS techniques.



You can read more about using Bower at the application's [website](#).

Bower can be used with other tools, such as Grunt, demonstrated later in [Recipe 12.14](#).

12.12. Compiling Node.js Modules for Use in the Browser with Browserify

Problem

Node has a lot of really great modules that you'd really like to use in your browser.

Solution

You can use Browserify to compile the Node module into browser accessible code. If it's one of the Node core modules, many are already compiled into shims that can be used in your browser application.

For instance, if you're interested in using the Node `querystring` module functionality, you create a client JavaScript bundle using the following Browserify command:

```
browserify -r querystring > bundle.js
```

Then use the module in your browser app:

```
<script src="bundle.js" type="text/javascript">
</script>
<script type="text/javascript">
  var qs = require('querystring');

  var str = qs.stringify({ first: 'apple', second: 'pear', third: 'pineapple' })
;
  console.log(str); //first=apple&second=pear&third=pineapple
</script>
```

Discussion

Browserify is a tool that basically moves Node functionality to the browser, as long as doing so makes sense. Of course, some functionality won't work (think input/output) but a surprising amount of functionality, including that in Node core, can work in the browser.

Browserify is installed via npm:

```
npm install -g browserify
```

It runs at the command line, as shown in the solution. In the solution, the `-r` flag triggers Browserify into creating a `require()` function to wrap the module's functionality, so we can use it in a similar manner in the browser app. The `querystring` module is one of the many Node core modules already compiled as a shim. The others are:

- `assert`
- `buffer`

- console
- constants
- crypto
- domain
- events
- http
- https
- os
- path
- punycode
- querystring
- stream
- string_decoder
- timers
- tty
- url
- util
- vm
- zlib

You can also compile other Node modules into browser code, including your own. As an example, let's say I have the following three Node files:

one.js

```
module.exports = function() {  
  console.log('hi from one');  
};
```

two.js

```
var one = require ('./one');  
  
module.exports = function(val) {  
  one();  
  console.log('hi ' + val + ' from two');  
};
```

index.js


```

var two = require ('./two');

module.exports = function() {
  two('world');
  console.log("And that's all");
}

```

I compiled it into an *appl.js* file using the following:

```
browserify ./index.js -o ./appl.js
```

Including the library in a web page results in the same three console `log()` function calls as you would see if you ran the original *index.js* file with Node, as soon as the generated script file is loaded.

12.13. Unit Testing Your Node Modules

Problem

You want to know the best way to ensure your module is ready for others to try.

Solution

Add *unit tests* as part of your production process.

Given the following module, named *bbarry*, and created in a file named *index.js* in the module directory:

```

var util = require('util');

(function(global) {
  'use strict';

  var bbarry = {};

  bbarry.concatArray = function (str, array) {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    } else if (typeof str !== 'string') {
      return -1;
    } else {
      return array.map(function(element) {
        return str + ' ' + element;
      });
    }
  };

  bbarry.splitArray = function (str,array) {
    if (!util.isArray(array) || array.length === 0) {
      return -1;
    } else if (typeof str !== 'string') {
      return -1;
    }
  };
}

```

```

    } else {
      return array.map(function(element) {
        var len = str.length + 1;
        return element.substring(len);
      });
    }
  };
  if (typeof module !== 'undefined' && module.exports) {
    module.exports = bbarry;
  } else if (typeof define === "function" && define.amd) {
    define("bbarry", [], function() {
      return bbarry;
    });
  } else {
    global.bbarry = bbarry;
  }
})(this));

```

Using Mocha, a JavaScript testing framework, and Node's built-in `assert` module, the following unit test (created as *index.js* and located in the project's *test* subdirectory) should result in the successful pass of six tests:

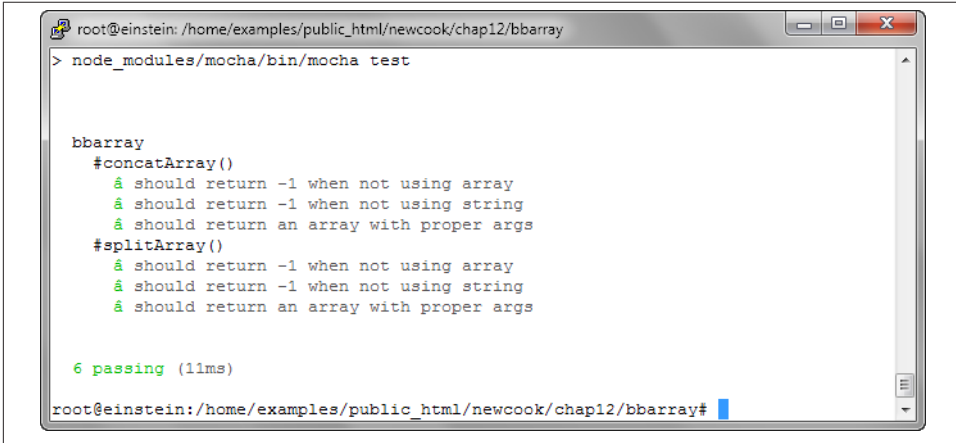
```

var assert = require('assert');
var bbarry = require('../index.js');

describe('bbarry', function() {
  describe('#concatArray()', function() {
    it('should return -1 when not using array', function() {
      assert.equal(-1, bbarry.concatArray(9, 'str'));
    });
    it('should return -1 when not using string', function() {
      assert.equal(-1, bbarry.concatArray(9, ['test', 'two']));
    });
    it('should return an array with proper args', function() {
      assert.deepEqual(['is test', 'is three'],
        bbarry.concatArray('is', ['test', 'three']));
    });
  });
  describe('#splitArray()', function() {
    it('should return -1 when not using array', function() {
      assert.equal(-1, bbarry.splitArray(9, 'str'));
    });
    it('should return -1 when not using string', function() {
      assert.equal(-1, bbarry.splitArray(9, ['test', 'two']));
    });
    it('should return an array with proper args', function() {
      assert.deepEqual(['test', 'three'],
        bbarry.splitArray('is', ['is test', 'is three']));
    });
  });
});

```

The result of the test is shown in [Figure 12-2](#), run using `npm test`.

A terminal window titled 'root@einstein: /home/examples/public_html/newcook/chap12/bbarry' showing the execution of 'node_modules/mocha/bin/mocha test'. The output lists six tests for 'bbarry' functions: '#concatArray()' and '#splitArray()', each with three sub-tests. All tests pass, resulting in '6 passing (11ms)'. The prompt 'root@einstein: /home/examples/public_html/newcook/chap12/bbarry#' is visible at the bottom.

```
root@einstein: /home/examples/public_html/newcook/chap12/bbarry
> node_modules/mocha/bin/mocha test

bbarry
  #concatArray()
    â should return -1 when not using array
    â should return -1 when not using string
    â should return an array with proper args
  #splitArray()
    â should return -1 when not using array
    â should return -1 when not using string
    â should return an array with proper args

6 passing (11ms)

root@einstein: /home/examples/public_html/newcook/chap12/bbarry#
```

Figure 12-2. Running unit tests based on Node Assert and Mocha

Discussion

Unit testing is one of those development tasks that may seem like a pain when you first start, but can soon become second nature. I don't necessarily agree with the folks that believe we should write the unit tests (*test-driven development*) first, before writing the code. But developing both test and code in parallel to each other should be a goal.

A *unit test* is a way that developers test their code to ensure it meets the specifications. It involves testing functional behavior, and seeing what happens when you send bad arguments—or no arguments at all. It's called unit testing because it's used with individual units of code, such as testing one module in a Node application, as compared to testing the entire Node application. It becomes one part of *integration testing*, where all the pieces are plugged together, before going to *user acceptance testing*: testing to ensure that the application does what users expect it to do (and that they generally don't hate it when they use it).

In the solution, I use two different functionalities for testing: Node's built-in `assert` module, and Mocha, a sophisticated testing framework. My module is simple, so I'm not using some of the more complex Mocha testing mechanisms. However, I think you'll get a feel for what's happening.

To install Mocha, use the following:

```
npm install mocha --save-dep
```

I'm using the `--save-dep` flag, because I'm installing Mocha into the module's Node dependencies. In addition, I modify the module's *package.json* file to add the following section:

```
"scripts": {  
  "test": "node_modules/mocha/bin/mocha test"  
},
```

The test script is saved as *index.js* in the *test* subdirectory under the project. The following command runs the test:

```
npm test
```

The Mocha unit test makes use of assertion tests from Node's `assert` module.

12.14. Running Tasks with Grunt

Problem

Pulling your Node module together is getting more complex—too complex to manually manage all of the elements.

Solution

Use a task runner like Grunt to manage all the bits for you.

For the following *barray* module:

```
var util = require('util');  
  
(function(global) {  
  'use strict';  
  
  var barray = {};  
  
  barray.concatArray = function (str, array) {  
    if (!util.isArray(array) || array.length === 0) {  
      return -1;  
    } else if (typeof str !== 'string') {  
      return -1;  
    } else {  
      return array.map(function(element) {  
        return str + ' ' + element;  
      });  
    }  
  };  
  
  barray.splitArray = function (str,array) {  
    if (!util.isArray(array) || array.length === 0) {  
      return -1;  
    } else if (typeof str !== 'string') {
```

```

        return -1;
    } else {
        return array.map(function(element) {
            var len = str.length + 1;
            return element.substring(len);
        });
    }
};

if (typeof module !== 'undefined' && module.exports) {
    module.exports = bbararray;
} else if (typeof define === 'function' && define.amd) {
    define('bbararray', [], function() {
        return bbararray;
    });
} else {
    global.bbararray = bbaarray;
}

})(this));

```

Saved as *bbararray.js* in the root directory, with a Mocha test file:

```

var assert = require('assert');
var bbararray = require('../bbararray.js');

describe('bbararray', function() {
    describe('#concatArray()', function() {
        it('should return -1 when not using array', function() {
            assert.equal(-1, bbararray.concatArray(9, 'str'));
        });
        it('should return -1 when not using string', function() {
            assert.equal(-1, bbararray.concatArray(9, ['test', 'two']));
        });
        it('should return an array with proper args', function() {
            assert.deepEqual(['is test', 'is three'],
                bbararray.concatArray('is', ['test', 'three']));
        });
    });
    describe('#splitArray()', function() {
        it('should return -1 when not using array', function() {
            assert.equal(-1, bbararray.splitArray(9, 'str'));
        });
        it('should return -1 when not using string', function() {
            assert.equal(-1, bbararray.splitArray(9, ['test', 'two']));
        });
        it('should return an array with proper args', function() {
            assert.deepEqual(['test', 'three'],
                bbararray.splitArray('is', ['is test', 'is three']));
        });
    });
});

```

Saved as *index.js* in a *test* subdirectory, the Grunt file is:

```
module.exports = function(grunt) {
  var banner = '/*\n<%= pkg.name %> <%= pkg.version %>';
  banner += '- <%= pkg.description %>\n<%= pkg.repository.url %>\n';
  banner += 'Built on <%= grunt.template.today("yyyy-mm-dd") %>\n*/\n';

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    jshint: {
      files: ['gruntfile.js', 'src/*.js'],
      options: {
        maxlen: 80,
        quotmark: 'single'
      }
    },
    uglify: {
      options: {
        banner: banner,
      },
      build: {
        files: {
          'build/<%= pkg.name %>.min.js':
            ['build/<%= pkg.name %>.js'],
        }
      }
    },
    simplemocha: {
      options: {
        globals: ['assert'],
        timeout: 3000,
        ignoreLeaks: false,
        ui: 'bdd',
        reporter: 'tap'
      },
      all: { src: ['test/*.js'] }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-simple-mocha');

  grunt.registerTask('default',
    ['jshint', 'simplemocha', 'uglify']);
};
```

When the file is saved as *gruntfile.js*, Grunt runs all the tasks defined in the file:

```
grunt
```

Discussion

Grunt is a *task runner*. Its only purpose is to consistently run a series of tasks. It's similar to the old Makefile, but without the decades of musty history.

To use Grunt, install it first:

```
npm install -g grunt-cli
```

Grunt needs to run in the same directory as your application/module's *package.json* file, as it works with the file. You can create either a JavaScript or Coffee-based Grunt file, but I'm focusing on the JS version.

Create the file by using the `grunt-init` CLI, with a given template, or you can use the example file given in the [Getting Started Guide](#).

A module needs to run within a certain framework to work with Grunt. Luckily, plugins have been created for many of the commonly used modules, such as the plugins used in the example for JSHint, Uglify, and Mocha. To ensure they're listed in the *package.json* file, they need to be installed using `--save-dev`:

```
npm install grunt-contrib-jshint --save-dev
npm install grunt-simple-mocha --save-dev
npm install grunt-contrib-uglify --save-dev
```

Each plugin also provides instructions about how to modify the Gruntfile to use the plugin and process your files.

Once you have both the *package.json* and *gruntfile.js* files running, the following will install any of the dependencies in the file, and run the Grunt tasks:

```
npm install
grunt
```

The result of running Grunt with the file in the solution is:

```
Running "jshint:files" (jshint) task
>> 1 file lint free.

Running "simplemocha:all" (simplemocha) task
1..6
ok 1 bbaray concatArray() should return -1 when not using array
ok 2 bbaray concatArray() should return -1 when not using string
ok 3 bbaray concatArray() should return an array with proper args
ok 4 bbaray splitArray() should return -1 when not using array
ok 5 bbaray splitArray() should return -1 when not using string
ok 6 bbaray splitArray() should return an array with proper args
# tests 6
# pass 6
# fail 0

Running "uglify:build" (uglify) task
>> Destination build/bbaray.min.js not written because src files were empty.
```

Done, without errors.

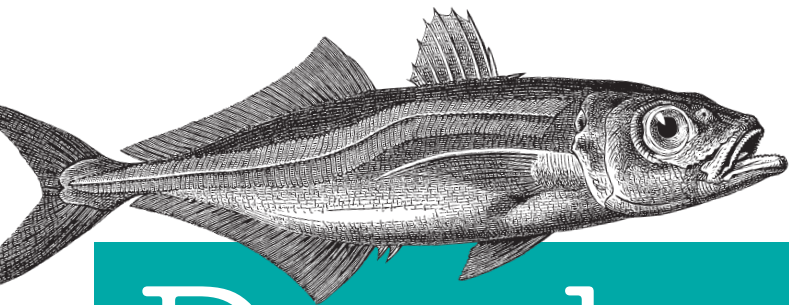
There are no files in the *src* directory, but I left the instructions in the Grunt file, for future expansion of the module.

See Also

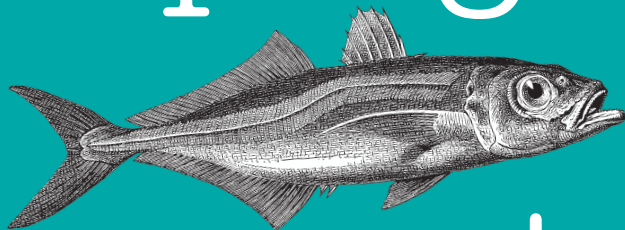
Read all about Grunt, and check out the available plugins, at the [application's website](#).

Another popular build system is [Gulp](#).

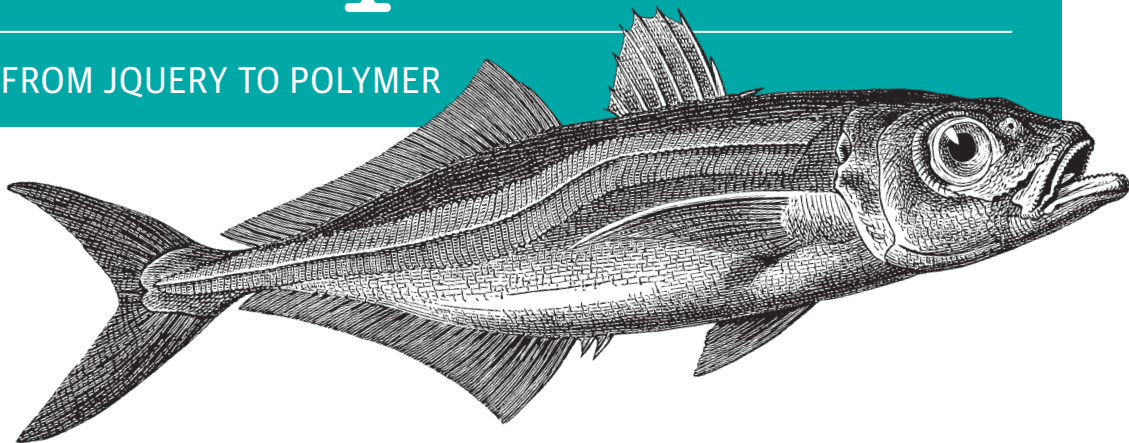
O'REILLY®



Developing Web Components



UI FROM JQUERY TO POLYMER



Jarrold Overson &
Jason Strimpel

Developing Web Components

Jarrold Overson and Jason Strimpel

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Working with the Shadow DOM

Jason Strimpel

The shadow DOM is not the dark side of the DOM, but if it were I would definitely give in to my hatred of the lack of encapsulation the DOM normally affords and cross over.

One of the aspects of the DOM that makes development of widgets/components difficult is this lack of encapsulation. For instance, one major problem has always been CSS rules bleeding into or out of a component's branch of the DOM tree: it forces one to write overly specific selectors or abuse `!important` so that styles do not conflict, and even then conflicts still happen in large applications. Another issue caused by lack of encapsulation is that code external to a component can still traverse into the component's branch of the DOM tree. These problems and others can be prevented by using the shadow DOM.

What Is the Shadow DOM?

So what exactly is this mysterious-sounding shadow DOM? According to [the W3C](#):

Shadow DOM is an adjunct tree of DOM nodes. These shadow DOM subtrees can be associated with an element, but do not appear as child nodes of the element. Instead the subtrees form their own scope. For example, a shadow DOM subtree can contain IDs and styles that overlap with IDs and styles in the document, but because the shadow DOM subtree (unlike the child node list) is separate from the document, the IDs and styles in the shadow DOM subtree do not clash with those in the document.



I am an admirer of the W3C, but oftentimes their specifications, albeit accurate, need to be translated into something that the rest of us—myself included—can more easily comprehend. The shadow DOM is essentially a way to define a new DOM tree whose root container, or host, is visible in the document, while the shadow root and its children are not. Think of it as a way to create isolated DOM trees to prevent collisions such as duplicate identifiers, or accidental modifications by broad query selectors. That is a simplification, but it should help to illustrate the purpose.

So what benefits does the shadow DOM provide to developers? It essentially provides encapsulation for a subtree from the parent page. This subtree can contain markup, CSS, JavaScript, or any asset that can be included in a web page. This allows you to create widgets without being concerned that any of the assets will impact the parent page, or vice versa. Previously this level of encapsulation was only achievable by using an `<iframe>`.

Shadow DOM Basics

The shadow DOM is a simple concept, but it has some intricacies that make it appear more complex than it really is. This section will focus on the basics. The intricacies that afford the developer even more control and power will be covered later in this chapter.

Shadow Host

A shadow host is a DOM node that contains a shadow root. It is a regular element node within the parent page that hosts the scoped shadow subtree. Any child nodes that reside under the shadow host are still selectable, with the exception of the shadow root.

Shadow Root

A shadow root is an element that gets added to a shadow host. The shadow root is the root node for the shadow DOM branch. Shadow root child nodes are not returned by DOM queries even if a child node matches the given query selector. Creating a shadow root on a node in the parent page makes the node upon which it was created a shadow host.

Creating a shadow root

Creating a shadow root is a straightforward process. First a shadow host node is selected, and then a shadow root is created in the shadow host.



To inspect shadow DOM branches using the Chrome debugger, check the “Show Shadow DOM” box under the “Elements” section in the “General” settings panel of the debugger.

The code to create a shadow root looks like this:

```
<div id="host"></div>

var host = document.querySelector('#host');
var root = host.createShadowRoot();
```



If you do not prefix `createShadowRoot` with “webkit” in Chrome 34 and below you are going to have a **bad time**. All calls to `createShadowRoot` should look like `host.webkitCreateShadowRoot()`.

It is possible to attach multiple shadow roots to a single shadow host. However, only the last shadow root attached is rendered. A shadow host follows the LIFO pattern (last in, first out) when attaching shadow roots. At this point you might be asking yourself, “So what is the point of hosting multiple shadow roots if only the last one attached is rendered?” Excellent question, but you are getting ahead of the game! This will be covered later in this chapter (see “[Shadow Insertion Points](#)” on page 120).

Using a Template with the Shadow DOM

Using a template to populate a shadow root involves almost the same process as using a template to add content to a DOM node in the parent page. The only difference is that the `template.content` is added to the shadow root.

The first step is to create a template node. This example leverages the template from the previous chapter, with the addition of an element that will be the shadow host:

```
<head>
  <template id="atcq">
    <p class="response"></p>
    <script type="text/javascript">
      (function () {
        var p = confirm('You on point Tip?');
        var responseEl = document.querySelector('#atcq-root')
          .shadowRoot
          .querySelector('.response');

        if (p) {
          responseEl.innerHTML = 'All the time Phife';
        } else {
          responseEl.innerHTML = 'Check the rhyme y\'all';
        }
      })();
    </script>
  </template>
</head>
```

```

    }
  })();
</script>
</template>
</head>
<body>
  <div id="atcq-root"></div>
</body>

```

Next, we create a shadow root using the shadow host element, get a reference to the template node, and finally append the template content to the shadow root:

```

// create a shadow root
var root = document.querySelector('#atcq-root').createShadowRoot();
// get a reference to the template node
var template = document.querySelector('#atcq');
// append the cloned content to the shadow root
root.appendChild(template.content);

```

Shadow DOM Styling

I cannot count the number of times I have encountered CSS scoping issues throughout my career. Some of them were due to broad selectors such as `div`, the overusage of `!important`, or improperly namespaced CSS. Other times it has been difficult to override widget CSS or widget CSS has bled out, impacting application-level CSS. As an application grows in size, especially if multiple developers are working on the code base, it becomes even more difficult to prevent these problems. Good standards can help to mitigate these issues, but most applications leverage open source libraries such as jQuery UI, Kendo UI, Bootstrap, and others, which makes good standards alone inadequate. Addressing these problems and providing a standard way of applying styles to scoped elements are two of the benefits of using a shadow DOM.

Style Encapsulation

Any styles defined in the shadow DOM are scoped to the shadow root. They are not applied to any elements outside of this scope, even if their selector matches an element in the parent page. Styles defined outside of a shadow DOM are not applied to elements in the shadow root either.

In the example that follows, the text within the `<p>` that resides outside of the shadow root will be blue because that style is external to the shadow DOM that is created. The text within the `<p>` that is a child node of the shadow root will initially be the default color. This is because the styles defined outside of the shadow root are not applied to elements within the shadow root. After two seconds, the text within the `<p>` inside the shadow root will turn green, because the callback for the `setTimeout` function injects a `<style>` tag into the shadow root. The text within the `<p>` that resides outside of the shadow root will remain blue because the style injected into the shadow

root is scoped to elements that are children of the shadow root. Here's the code that achieves this styling:

```
<head>
  <style>
    p {
      color: blue;
    }
  </style>
  <template><p>I am the default color, then green.</p></template>
</head>
<body>
  <div id="host"></div>
  <p>I am blue.</p>
  <script type="text/javascript">
    var template = document.querySelector('template');
    var root = document.querySelector('#host').createShadowRoot();

    root.appendChild(template.content);
    setTimeout(function () {
      root.innerHTML += '<style>p { color: green; }</style>';
    }, 2000)
  </script>
</body>
```

Styling the Host Element

In some cases you will want to style the host element itself. This is easily accomplished by creating a style anywhere within the parent page, because the host element is not part of the shadow root. This works fine, but what if you have a shadow host that needs different styling depending on the contents of the shadow root? And what if you have multiple shadow hosts that need to be styled based on their contents? As you can imagine, this would get very difficult to maintain. Fortunately, there is a new selector, `:host`, that provides access to the shadow host from within the shadow root. This allows you to encapsulate your host styling to the shadow root:

```
<head>
  <template id="template">
    <style>
      :host {
        border: 1px solid red;
        padding: 10px;
      }
    </style>
    My host element will have a red border!
  </template>
</head>
<body>
  <div id="host"></div>
  <script type="text/javascript">
    var template = document.querySelector('#template')
```

```

        var root = document.querySelector('#host').createShadowRoot();
        root.appendChild(template.content);
    </script>
</body>

```

The parent page selector has a higher specificity than the `:host` selector, so it will trump any shadow host styles defined within the shadow root:

```

<head>
  <style>
    #host {
      border: 1px solid green;
    }
  </style>
  <template id="template">
    <style>
      :host {
        border: 1px solid red;
        padding: 10px;
      }
    </style>
    My host element will have a green border!
  </template>
</head>
<body>
  <div id="host"></div>
  <script type="text/javascript">
    var template = document.querySelector('#template')
    var root = document.querySelector('#host').createShadowRoot();
    root.appendChild(template.content);
  </script>
</body>

```

If you want to override styles set in the parent page, this must be done inline on the host element:

```

<head>
  <template id="template">
    <style>
      :host {
        border: 1px solid red;
        padding: 10px;
      }
    </style>
    My host element will have a blue border!
  </template>
</head>
<body>
  <div id="host" style="border: 1px solid blue;"></div>
  <script type="text/javascript">
    var template = document.querySelector('#template')
    var root = document.querySelector('#host').createShadowRoot();
    root.appendChild(template.content);
  </script>
</body>

```



```
</script>
</body>
```

The `:host` selector also has a functional form that accepts a selector, `:hostselector`, allowing you to set styles for specific hosts. This functionality is useful for theming and managing states on the host element.

Styling Shadow Root Elements from the Parent Page

Encapsulation is all well and good, but what if you want to target specific shadow root elements with a styling update? What if you want to reuse templates and shadow host elements in a completely different application? What if you do not have control over the shadow root's content? For instance, you could be pulling the code from a repository that is maintained by another department internal to your organization, or a shared repository that is maintained by an external entity. In either case you might not have control over the shadow root's contents, or the update process might take a significant amount of time, which would block your development. Additionally, you might not *want* control over the content. Sometimes it is best to let domain experts maintain certain modules and to simply override the default module styling to suit your needs. Fortunately, the drafters of the W3C specification thought of these cases (and probably many more), so they created a selector that allows you to apply styling to shadow root elements from the parent page.

The `::shadow` pseudoelement selects the shadow root, allowing you to target child elements within the selected shadow root:

```
<head>
  <style>
    #host::shadow p {
      color: blue;
    }
  </style>
  <template><p>I am blue.</p></template>
</head>
<body>
  <div id="host"></div>
  <script type="text/javascript">
    var template = document.querySelector('template');
    var root = document.querySelector('#host').createShadowRoot();

    root.appendChild(template.content);
  </script>
</body>
```

The `::shadow` pseudoelement selector can be used to style nested shadow roots:

```
<head>
  <style>
    #parent-host::shadow #child-host::shadow p {
      color: blue;
    }
  </style>
</head>
```

```

    }
</style>
<template id="child-template"><p>I am blue.</p></template>
<template id="parent-template">
  <p>I am the default color.</p>
  <div id="child-host"></div>
</template>
</head>
<body>
  <div id="parent-host"></div>
  <script type="text/javascript">
    var parentTemplate = document.querySelector('#parent-template');
    var childTemplate = document.querySelector('#child-template');
    var parentRoot = document.querySelector('#parent-host')
      .createShadowRoot();
    var childRoot;

    parentRoot.appendChild(parentTemplate.content);
    childRoot = parentRoot.querySelector('#child-host').createShadowRoot();
    childRoot.appendChild(childTemplate.content);
  </script>
</body>

```

Sometimes targeting individual shadow roots using the `::shadow` pseudoelement is very inefficient, especially if you are applying a theme to an entire application of shadow roots. Again, the drafters of the W3C specification had the foresight to anticipate this use case and specified the `/deep/` combinator. The `/deep/` combinator allows you cross through all shadow roots with a single selector:

```

<style>
  /* colors all <p> text within all shadow roots blue */
  body /deep/ p {
    color: blue;
  }

  /* colors all <p> text within the child shadow root blue */
  #parent-host /deep/ #child-host # p {
    color: blue;
  }

  /* targets a library theme/skin */
  body /deep/ p.skin {
    color: blue;
  }
</style>

```



At this point you might be asking yourself, “Doesn’t this defeat the purpose of encapsulation?” But encapsulation does not mean putting up an impenetrable force field that makes crossing boundaries for appropriate use cases, such as theming UI components, impossible. The problem with the Web is that it has never had a formalized method of encapsulation or a defined API for breaking through an encapsulated component, like in other development platforms. The formalization of encapsulation and associated methods makes it clear in the code what the developer’s intent is when encapsulation is breached. It also helps to prevent the bugs that plague a web platform that lacks formalized encapsulation.

Content Projection

One of the main tenets of web development best practices is the separation of content from presentation, the rationale being that it makes application maintenance easier and more accessible.

In the past separation of content from presentation has simply meant not placing styling details in markup. The shadow DOM takes this principle one step further.

In the examples we have seen thus far the content has been contained within a template and injected into the shadow root. In these examples no significant changes were made to the presentation, other than the text color. Most cases are not this simple.

In some cases it is necessary to place the content inside of the shadow host element for maintenance and accessibility purposes. However, that content needs to be projected into the shadow root in order to be presented. Luckily, the building blocks for projecting the content from the shadow host into the shadow root exist.

Projection via a Content Tag

One way to project content from the shadow host into the shadow root is by using a `<content>` element inside of a `<template>`. Any content inside the shadow host will be automatically projected into the `<content>` of the `<template>` used to compose the shadow root:

```
<head>
  <meta charset="utf-8">
  <title>Test Code</title>
  <template>
    <p>I am NOT projected content.</p>
    <content></content>
  </template>
</head>
<body>
```

```

<div id="host">
  <p>I am projected content.</p>
</div>
<script type="text/javascript">
  var template = document.querySelector('template');
  var root = document.querySelector('#host').createShadowRoot();

  root.appendChild(template.content);
</script>
</body>

```

Projection via Content Selectors

In some cases you may not want to project all of the content from the shadow host. You might want to select specific content for injection in different `<content>` elements in the shadow root. A common case is that some markup in the shadow host has semantic meaning and helps with accessibility, but doesn't add anything to the presentation of the shadow root. The mechanism for extracting content from the shadow host is the `select` attribute. This attribute can be added to a `<content>` element with a query selector value that will match an element in the shadow host. The matched element's content is then injected into the `<content>` tag.



Only the first element matched by a `<content>` element's `select` attribute is injected into the element—keep this in mind when using selectors that are likely to match a number of elements, such as tag selectors (e.g., `div`).

The following example is a product listing with a review widget. The shadow host contains semantic markup that is accessible, and the template contains the presentation details. The template presentation does not lend itself well to accessibility and contains extraneous markup that is used for presentation purposes only—e.g., the column containers are there for positioning purposes only and the review `` items do not contain text (the interface would be purely graphical). In this example, `select` attributes are used in the `<template>` `<content>` elements to extract content from the shadow root:

```

<!--
  Only the relevant markup is shown. All other details,
  such as template CSS and JavaScript, have been omitted,
  so that the focus is on the selector projection use case.
-->
<template>
  <div class="product">
    <div class="column main">
      <content select="h2"></content>
      <content select=".description"></content>
    </div>

```

```

<div class="column sidebar">
  <content select="h3"></content>
  <ul class="ratings">
    <li class="1-star"></li>
    <li class="2-star"></li>
    <li class="3-star"></li>
    <li class="4-star"></li>
  </ul>
</div>
</div>
</template>
<div id="host" class="product">
  <h2>ShamWow</h2>
  <p class="description">
    ShamWow washes, dries, and polishes any surface. It's like a towel,
    chamois, and sponge all in one!
  </p>
  <h3>Ratings</h3>
  <ul class="ratings">
    <li>1 star</li>
    <li>2 stars</li>
    <li>3 stars</li>
    <li>4 stars</li>
  </ul>
</div>

```



Only nodes that are children of the shadow host can be projected, so you cannot select content from any lower descendant in the shadow host.

Getting Distributed Nodes and Insertion Points

Nodes that are projected from a host are referred to as *distributed nodes*. These nodes do not actually move locations in the DOM, which makes sense because the same host child node can be projected to different insertion points across shadow roots. As you can imagine, things can get complicated rather quickly, and at times you may need to do some inspecting and take action on distributed nodes in your application. There are two different methods that support this, `Element.getDistributedNodes` and `Element.getDestinationInsertionPoints`.



You cannot traverse into a `<content>` tree, because a `<content>` node does not have any descendant nodes. It is helpful to think of a `<content>` node as a television that is displaying a program. The television's only role in producing the program is to display it. The program itself was filmed and edited elsewhere for consumption by an unlimited number of televisions.

We use these methods as follows:

```
// Element.getDistributedNodes
var root = document.querySelector('#some-host').createShadowRoot();

// iterate over all the content nodes in the root
[].forEach.call(root.querySelectorAll('content'), function (contentNode) {
  // get the distributed nodes for each content node
  // and iterate over the distributed nodes
  [].forEach.call(contentNode.getDistributedNodes(),
    function (distributedNode) {
      // do something cool with the contentNode
    });
});

// Element.getDestinationInsertionPoints
var hostChildNode = document.querySelector('#some-host .some-child-node');

// get child node insertion points and iterate over them
[].forEach.call(hostChildNode.getDestinationInsertionPoints(),
  function (contentNode) {
    // do something cool with the contentNode
  });
```

Shadow Insertion Points

In the previous section we examined how shadow host content can be projected into insertion points, `<content>`. Just as content can be projected into insertion points, so can shadow roots. Shadow root insertion points are defined using `<shadow>` tags. Like any other tags, these can be created directly in markup or added to the DOM programmatically.



Shadow roots are stacked in the order they are added, with the youngest shadow root tree appearing last and rendering. Trees appearing earlier in the stack are referred to as *older trees*, while trees appearing after a given shadow root are referred to as *younger trees*.

At the beginning of this chapter it was stated that a shadow host could contain multiple shadow roots, but that only the last shadow root defined would be rendered. This is true in the absence of `<shadow>` elements. The `<shadow>` tag provides a point for projecting an older shadow root using a younger shadow root tree. If a shadow root tree contains more than one `<shadow>` element, the first one is used and the rest are ignored. Essentially, `<shadow>` allows you to render an older shadow root in a stack by providing an insertion point for it to be projected into.



Projecting nodes to an insertion point does not affect the tree structure. The projected nodes remain in their original locations within the tree. They are simply displayed in the assigned insertion point.

Here's an example:

```
<template id="t-1">I am t1. </template>
<template id="t-2"><shadow></shadow>I am t2. </template>
<template id="t-3"><shadow></shadow>I am t3. </template>
<div id="root"></div>
<script type="text/javascript">
  (function () {
    var t1 = document.querySelector('#t-1');
    var t2 = document.querySelector('#t-2');
    var t3 = document.querySelector('#t-3');
    var host = document.querySelector('#root');
    var r1 = host.createShadowRoot();
    var r2 = host.createShadowRoot();
    var r3 = host.createShadowRoot();

    r1.appendChild(t1.content);
    r2.appendChild(t2.content);
    r3.appendChild(t3.content);
  })();
</script>
<!-- renders: "I am t1. I am t2. I am t3." -->
```

The previous code block renders from the bottom (youngest tree) up, projecting the next-oldest shadow root tree into the `<shadow>` insertion point. This was a very simple example. In a real application, the code will be more complicated and dynamic. Because of this it is helpful to have a way to inspect a `<shadow>` programmatically or to determine a shadow host's root:

```
// using the previous code block as an example
// determine older shadow root
r1.olderShadowRoot === null; // true; first in the stack
r2.olderShadowRoot === r1; // true
r3.olderShadowRoot === r2; // true
```

```
// determine a host's shadow root
host.shadowRoot === r1; // false; there can only be one (LIFO)
host.shadowRoot === r2; // false; ditto
host.shadowRoot === r3; // true

// determine a shadow root's host
r1.host === host; // true
r2.host === host; // true
r3.host === host; // true
```

Events and the Shadow DOM

At this point you might be thinking that projecting nodes instead of cloning them is a great optimization that will help to keep changes synchronized—but what about events bound to these projected nodes? How exactly does this work if they are not copied? In order to normalize these events, they are sometimes **retargeted** to appear as if they were triggered by the host element rather than the projected element in the shadow root. In these cases you can still determine the shadow root of the projected node by examining the `path` property of the event object. Some events are never retargeted, though, which makes sense if you think about it. For instance, how would a `scroll` event be retargeted? If a user scrolls one projected node, should the others scroll? The events that are not retargeted are:

- `abort`
- `error`
- `select`
- `change`
- `load`
- `reset`
- `resize`
- `scroll`
- `selectstart`

Updating the Dialog Template to Use the Shadow DOM

You might have noticed generic `id` values such as `title` and `content` were used in the dialog component, and you probably thought, “This idiot is going to have duplicate `id` values, which are supposed to be unique in the DOM, colliding left and right...” This was intentional, though, and has been leading up to this moment!

One of the benefits of the shadow DOM is the encapsulation of markup, which means the encapsulation of id values and a decrease in the likelihood of id value collisions in your application.

This code will leverage the previous chapter's code that demonstrated using a template to make the dialog component markup and JavaScript inert until it was appended to the DOM.

Dialog Markup

The dialog component will utilize a template, like the previous example, but this template will be appended to a shadow root that is hosted by `<div id="dialog-host">`. The interesting part about this example is that it is practically the reverse of our review widget example in terms of accessibility and readability. The `aria` attributes are contained within the shadow DOM, and the markup a developer writes is not exactly semantic. However, if you think about it, the `aria` attributes are primarily used for accessibility implementation details, so it makes sense that these details are obfuscated from the developer. The part that does not make sense is that the host markup is not very semantic, but please reserve judgment on that until the next chapter!

Here's the code for our updated dialog template:

```
<head>
  <script type="text/javascript" src="/vendor/jquery.js"></script>
  <template id="dialog">
    <style>
      // styling src
    </style>
    <script type="text/javascript">
      // dialog component source
    </script>
    <div role="dialog" aria-labelledby="title" aria-describedby="content">
      <h2 id="title"></h2>
      <p id="content"></p>
    </div>
  </template>
</head>
<!-- example host node -->
<div id="dialog-host">
  <h2>I am a title</h2>
  <p>Look at me! I am content.</p>
</div>
```

Dialog API

If you want to encapsulate the creation of the shadow root, the cloning and appending of the template, and the dialog component instantiation, then it is best to create a wrapper constructor function that encapsulates all of these implementation details:

```
function DialogShadow(options) {
  this.options = options;
  // get the host node using the hostQrySelector option
  this.host = document.querySelector(options.hostQrySelector);
  // grab the template
  this.template = document.querySelector('#dialog');
  this.root = this.host.createShadowRoot();
  // append the template content to the root
  this.root.appendChild(this.template.content);
  // get a reference to the dialog container element in the root
  this.el = this.root.querySelector('[role="dialog"]');

  this.options.$el = this.el;
  // align element to body since it is a fragment
  this.options.alignToEl = document.body;
  this.options.align = 'M';
  // do not clone node
  this.options.clone = false;

  // get the content from the host node; projecting would retain host
  // node styles and not allow for encapsulation of template styles
  this.el.querySelector('#title').innerHTML = this.host.querySelector('h2')
    .innerHTML;
  this.el.querySelector('#content').innerHTML = this.host.querySelector('p')
    .innerHTML;

  // create a dialog component instance
  this.api = new Dialog(this.options);

  return this;
}
```

Updating the Dialog show Method

Since the shadow root and its children are a subtree that is not part of the parent document, we have to ensure that the host element's z-index value is modified so that it appears on the top of its stacking context, the <body>:

```
// see GitHub repo for full example
(function (window, $, Voltron, Duvet, Shamen, ApacheChief, jenga) {

  'use strict';

  // makes dialog visible in the UI
  Dialog.prototype.show = function () {
```

```

        // this will adjust the z-index, set the display property,
        // and position the dialog
        this.overlay.position();
        // bring the host element to the top of the stack
        jenga.bringToFront(this.$el[0].parentNode.host);
    };

})(window, jQuery, Voltron, Duvet, Shamen, ApacheChief, jenga);

```

Instantiating a Dialog Component Instance

The dialog component can now be instantiated just as before, but it will now be scoped to the shadow root:

```

var dialog = new DialogShadow({
  draggable: true,
  resizable: true,
  hostQrySelector: '#dialog-host'
});

dialog.api.show();

```

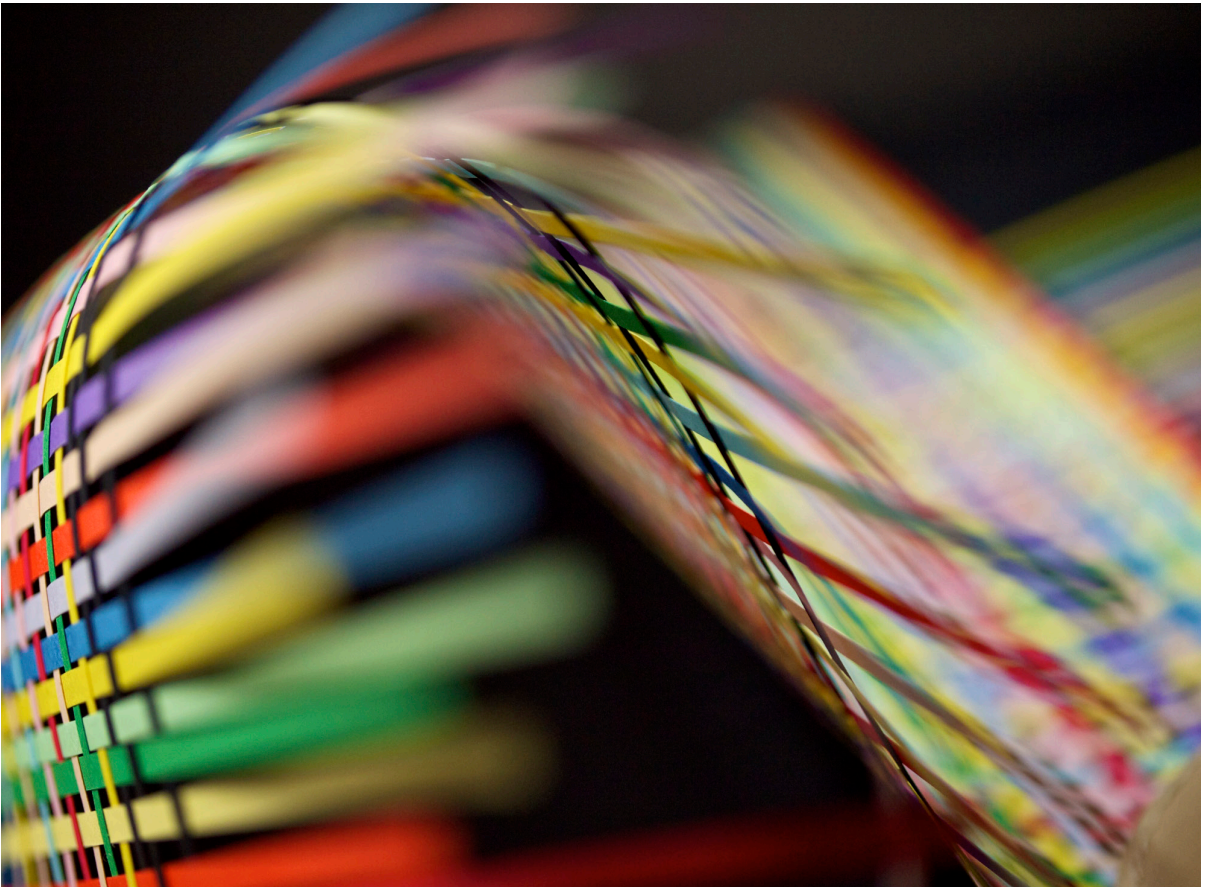
Summary

In this chapter we introduced the shadow DOM and discussed the primary benefit it affords developers: encapsulation. Before the shadow DOM, the only way to achieve this level of encapsulation was to use an `<i>iframe</i>`. We then discussed, in great detail, the encapsulation of styling, including the new supporting CSS selectors and the rationale for these new selectors. We then covered the projection of nodes to insertion points, using `<content>` and `<shadow>` elements. This included the usage of the new `select` attribute for selecting specific content from a host node. Next, we examined the properties and methods for inspecting distributed, host, and root nodes. After that, we highlighted how events work in host and root nodes. Finally, we updated the dialog component example to use a shadow DOM.

Beautiful JavaScript

Leading Programmers Explain
How They Think

Anton Kovalyov



Beautiful JavaScript

Edited by Anton Kovalyov

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Functional JavaScript

Anton Kovalyov

Is JavaScript a functional programming language? This question has long been a topic of great debate within our community. Given that JavaScript’s author was recruited to do “Scheme in the browser,” one could argue that JavaScript was designed to be used as a functional language. On the other hand, JavaScript’s syntax closely resembles Java-like object-oriented programming, so perhaps it should be utilized in a similar manner. Then there might be some other arguments and counterarguments, and the next thing you know the day is over and you didn’t do anything useful.

This chapter is not about functional programming for its own sake, nor is it about altering JavaScript to make it resemble a pure functional language. Instead, this chapter is about taking a pragmatic approach to functional programming in JavaScript, a method by which programmers use elements of functional programming to simplify their code and make it more robust.

Functional Programming

Programming languages come in several varieties. Languages like Go and C are called *procedural*: their main programming unit is the procedure. Languages like Java and SmallTalk are object oriented: their main programming unit is the object. Both these approaches are imperative, which means they rely on commands that act upon the machine state. An imperative program executes a sequence of commands that change the program’s internal state over and over again.

Functional programming languages, on the other hand, are oriented around expressions. Expressions—or rather, pure expressions—don’t have a state, as they merely compute a value. They don’t change the state of something outside their scope, and they don’t rely on data that can change outside their scope. As a result, you should be

able to substitute a pure expression with its value without changing the behavior of a program. Consider an example:

```
function add(a, b) {  
  return a + b  
}  
  
add(add(2, 3), add(4, 1)) // 10
```

To illustrate the process of substituting expressions, let's evaluate this example. We start with an expression that calls our `add` function three times:

```
add(add(2, 3), add(4, 1))
```

Since `add` doesn't depend on anything outside its scope, we can replace all calls to it with its contents. Let's replace the first argument that is not a primitive value—`add(2, 3)`:

```
add(2 + 3, add(4, 1))
```

Then we replace the second argument:

```
add(2 + 3, 4 + 1)
```

Finally, we replace the last remaining call to our function and calculate the result:

```
(2 + 3) + (4 + 1) // 10
```

This property that allows you to substitute expressions with their values is called *referential transparency*. It is one of the essential elements of functional programming.

Another important element of functional programming is *functions as first-class citizens*. Michael Fogus gave a great explanation of functions as first-class citizens in his book, *Functional JavaScript*. His definition is one of the best I've seen:

The term “first-class” means that something is just a value. A first-class function is one that can go anywhere that any other value can go—there are few to no restrictions. A number in JavaScript is surely a first-class thing, and therefore a first-class function has a similar nature:

- A number can be stored in a variable and so can a function:

```
var fortytwo = function() { return 42 };
```

- A number can be stored in an array slot and so can a function:

```
var fortytwos = [42, function() { return 42 }];
```

- A number can be stored in an object field and so can a function:

```
var fortytwos = {number: 42, fun: function() { return 42 }};
```

- A number can be created as needed and so can a function:

```
42 + (function() { return 42 })(); // => 84
```

- A number can be passed to a function and so can a function:

```
function weirdAdd(n, f) { return n + f() }

weirdAdd(42, function() { return 42 }); // => 84
```

- A number can be returned from a function and so can a function:

```
return 42;

return function() { return 42 };
```

Having functions as first-class citizens enables another important element of functional programming: higher-order functions. A *higher-order function* is a function that operates on other functions. In other words, higher-order functions can take other functions as their arguments, return new functions, or do both. One of the most basic examples is a higher-order `map` function:

```
map([1, 2, 3], function (n) { return n + 1 }) // [2, 3, 4]
```

This function takes two arguments: a collection of values and another function. Its result is a new list with the provided function applied to each element from the list.

Note how this `map` function uses all three elements of functional programming described previously. It doesn't change anything outside of its scope, nor does it use anything from the outside besides the values of its arguments. It also treats functions as first-class citizens by accepting a function as its second argument. And since it uses that argument to compute the value, one can definitely call it a higher-order function.

Other elements of functional programming include recursion, pattern matching, and infinite data structures, although I will not elaborate on these elements in this chapter.

Functional JavaScript

So, is JavaScript a truly functional programming language? The short answer is no. Without support for tail-call optimization, pattern matching, immutable data structures, and other fundamental elements of functional programming, JavaScript is not what is traditionally considered a truly functional language. One can certainly try to treat JavaScript as such, but in my opinion, such efforts are not only futile but also dangerous. To paraphrase Larry Paulson, author of the *Standard ML for the Working Programmer*, a programmer whose style is “almost” functional had better not be lulled into a false sense of referential transparency. This is especially important in a language like JavaScript, where one can modify and overwrite almost everything under the sun.

Consider `JSON.stringify`, a built-in function that takes an object as a parameter and returns its JSON representation:

```
JSON.stringify({ foo: "bar" }) // -> '{"foo":"bar"}'
```


One might think that this function is pure, that no matter how many times we call it or in what context we call it, it always returns the same result for the same arguments. But what if somewhere else, most probably in code you don't control, someone overwrites the `Object.prototype.toJSON` method?

```
JSON.stringify({ foo: "bar" })
// -> '{"foo":"bar"}'

Object.prototype.toJSON = function () {
  return "reality ain't always the truth"
}

JSON.stringify({ foo: "bar" })
// -> '"reality ain't always the truth'"
```

As you can see, by slightly modifying a built-in `Object`, we managed to change the behavior of a function that looks pretty pure and functional from the outside. Functions that read mutable references and properties aren't pure, and in JavaScript, most nontrivial functions do exactly that.

My point is that functional programming, especially when used with JavaScript, is about reducing the complexity of your programs and not about adhering to one particular ideology. Functional JavaScript is not about eliminating all the mutations; it's about reducing occurrences of such mutations and making them very explicit. Consider the following function, `merge`, which merges together two arrays by pairing their corresponding members:

```
function merge(a, b) {
  b.forEach(function (v, i) { a[i] = [a[i], b[i]] })
}
```

This particular implementation does the job just fine, but it also requires intimate knowledge of the function's behavior: does it modify the first argument, or the second?

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b)
a // -> [ [1, "one"], [2, "two"], .. ]
```

Imagine that you're unfamiliar with this function. You skim the code to review a patch, or maybe just to familiarize yourself with a new codebase. Without reading the function's source, you have no information regarding whether it merges the first argument into the second, or vice versa. It's also possible that the function is not destructive and someone simply forgot to use its value.

Alternatively, you can rewrite the same function in a nondestructive way. This makes the state change explicit to everyone who is going to use that function:

```
function merge(a, b) {
  return a.map(function (v, i) { return [v, b[i]] })
}
```

Since this new implementation doesn't modify any of its arguments, all mutations will have to be explicit:

```
var a = [1, 2, 3]
var b = ["one", "two", "three"]

merge(a, b) // -> [ [1, "one"], [2, "two"], .. ]

// a and b still have their original values.
// Any change to the value of a will have to
// be explicit through an assignment:
a = merge(a, b)
```

To further illustrate the difference between the two approaches, let's run that function three times without assigning its value:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
merge(a, b)
// -> [ [1, "one"], [2, "two"] ]; a and b are the same
```

As you can see, the return value never changes. It doesn't matter how many times you run this function; the same input will always lead to the same output. Now let's go back to our original implementation and perform the same test:

```
var a = [1, 2]
var b = ["one", "two"]

merge(a, b)
// -> undefined; a is now [ [1, "one"], [2, "two"] ]
merge(a, b)
// -> undefined; a is now [ [[1,"one"], "one"], [[2, "two"],"two"] ]
merge(a, b)
// -> undefined; a is even worse now; the universe imploded
```

Even better is that this version of `merge` allows us to use it *as a value itself*. We can return the result of its computation or pass it around without creating temporary variables, just like we would do with any other variable holding a primitive value:

```
function prettyTable(table) {
  return table.map(function (row) {
    return row.join(" ")
  }).join("\n")
}
```

```

console.log(prettyTable(merge([1, 2, 3], ["one", "two", "three"])))
// prints:
//   1 "one"
//   2 "two"
//   3 "three"

```

This type of function, known as a *zip* function, is quite popular in the functional programming world. It becomes useful when you have multiple data sources that are coordinated through matching array indexes. JavaScript libraries such as Underscore and LoDash provide implementations of *zip* and other useful helper functions so you don't have to reinvent the wheel within your projects.

Let's look at another example where explicit code reads better than implicit. JavaScript—at least, its newer revisions—allows you to create constants in addition to variables. Constants can be created with a `const` keyword. While everyone else (including yours truly) primarily uses this keyword to declare module-level constants, my friend Nick Fitzgerald uses `consts` virtually everywhere to make clear which variables are expected to be mutated and which are not:

```

function invertSourceMaps(code, mappings) {
  const generator = new SourceMapGenerator(...)

  return DevToolsUtils.yieldingEach(mappings, function (m) {
    // ...
  })
}

```

With this approach, you can be sure that a `generator` is always an instance of `SourceMapGenerator`, regardless of where it is being used. It doesn't give us immutable data structures, but it *does* make it impossible to point this variable to a new object. This means there's one less thing to keep track of while reading the code.

Here's a bigger example of a functional approach to programming: a few weeks ago, I wrote a static site generator in JavaScript for the [JSHint](#) website and my personal blog. The main module that actually reads all the templates, generates a new site, and writes it back to disk consists of only three small functions. The first function, `read`, takes a path as an argument and returns an object that contains the whole directory tree plus the contents of the source files. The second function, `build`, does all the heavy work: it compiles all the templates and Markdown files into HTML, compresses static files, and so on. The third function, `write`, takes the site structure and saves it to disk.

There's absolutely no shared state between those three functions. Each has a set of arguments it accepts and some data it returns. An executable script I use from my command line does precisely the following:

```
#!/usr/bin/env node

var oddweb = require("./index.js")
var args   = process.argv.slice(2)

oddweb.write(oddweb.build(oddweb.read(args[1])))
```

I also get plug-ins for free. If I need a plug-in that deletes all files with names ending with *.draft*, all I do is write a function that gets a site tree as an argument and returns a new site tree. I then plug in that function somewhere between `read` and `write`, and I'm golden.

Another benefit of using a functional programming style is simpler unit tests. A pure function takes in some data, computes it, and returns the result. This means that all that's needed in order to test that function is input data and an expected return value. As a simple example, here's a unit test for our function `merge`:

```
function testMerge() {
  var data = [
    { // Both lists have the same size
      a: [1, 2, 3],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"], [3, "c"] ]
    },

    { // Second list is larger
      a: [1, 2],
      b: ["a", "b", "c"],
      ret: [ [1, "a"], [2, "b"] ]
    },

    { // Etc.
      ...
    }
  ]

  data.forEach(function (test) {
    isEqual(merge(test.a, test.b), test.ret)
  })
}
```

This test is almost fully declarative. You can clearly see what input data is used and what is expected to be returned by the `merge` function. In addition, writing code in a functional way means you have less testing to do. Our original implementation of `merge` was modifying its arguments, so that a proper test would have had to cover cases where one of the arguments was frozen using `Object.freeze`.

All functions involved in the preceding example—`forEach`, `isEqual`, and `merge`—were designed to work with only simple, built-in data types. This approach, where you build your programs around composable functions that work with simple data types, is

called *data-driven programming*. It allows you to write programs that are clear and elegant and have a lot of flexibility for expansion.

Objects

Does this mean you shouldn't use objects, constructors, and prototype inheritance? Of course not! If something makes your code easier to understand and maintain, it'd be silly not to use it. However, JavaScript developers often start making overcomplicated object hierarchies without even considering whether there are simpler ways to solve the problem.

Consider the following object that represents a robot. This robot can walk and talk, but otherwise it's pretty useless:

```
function Robot(name) {  
  this.name = name  
}  
  
Robot.prototype = {  
  talk: function (what) { /* ... */ },  
  walk: function (where) { /* ... */ }  
}
```

What would you do if you wanted two more robots: a guard robot to shoot things and a housekeeping robot to clean things? Most people would immediately create child objects `GuardRobot` and `HousekeeperRobot` that inherit methods and properties from the parent `Robot` object and add their own methods. But what if you then decided you wanted a robot that can both clean and shoot things? This is where hierarchy gets complicated and software fragile.

Consider the alternative approach, where you extend instances with functions that define their behavior and not their type. You don't have a `GuardRobot` and a `HousekeeperRobot` anymore; instead, you have an instance of a `Robot` that can clean things, shoot things, or do both. The implementation will probably look something like this:

```
function extend(robot, skills) {  
  skills.forEach(function (skill) {  
    robot[skill.name] = skill.fn.bind(null, rb)  
  })  
  
  return robot  
}
```

To use it, all you have to do is to implement the behavior you need and attach it to the instance in question:

```
function shoot(robot) { /* ... */ }  
function clean(robot) { /* ... */ }
```

```
var rdo = new Robot("R. Daniel Olivaw")
extend(rdo, { shoot: shoot, clean: clean })

rdo.talk("Hi!") // OK
rdo.walk("Mozilla SF") // OK
rdo.shoot() // OK
rdo.clean() // OK
```

NOTE

My friend Irakli Gozalishvili, after reading this chapter, left a comment saying that his approach would be different. What if objects were used only to store data?

```
function talk(robot) { /* ... */ }
function shoot(robot) { /* ... */ }
function clean(robot) { /* ... */ }

var rdo = { name: "R. Daniel Olivaw" }

talk(rdo, "Hi!") // OK
walk(rdo, "Mozilla SF") // OK
shoot(rdo) // OK
clean(rdo) // OK
```

With his approach you don't even need to extend anything: all you need to do is pass the correct object.

At the beginning of this chapter, I warned JavaScript programmers against being lulled into the false sense of referential transparency that can result from using a pure functional programming language. In the example we just looked at, the function `extend` takes an object as its first argument, modifies it, and returns the modified object. The problem here is that JavaScript has a very limited set of immutable types. Strings are immutable. So are numbers. But objects—such as an instance of `Robot`—are mutable. This means that `extend` is not a pure function, since it mutates the object that was passed into it. You can call `extend` without assigning its return value to anything, and `rdo` will still be modified.

Now What?

The major evolution that is still going on for me is towards a more functional programming style, which involves unlearning a lot of old habits, and backing away from some OOP directions.

—John Carmack

JavaScript is a multiparadigm language supporting object-oriented, imperative, and functional programming styles. It provides a framework in which you can mix and match different styles and, as a result, write elegant programs. Some programmers, however, forget about all the different paradigms and stick only with their favorite

one. Sometimes this rigidity is due to fear of leaving a comfort zone; sometimes it's caused by relying too heavily on the wisdom of elders. Whatever the reason, these people often limit their options by confining themselves to a small space where it's their way or the highway.

Finding the right balance between different programming styles is hard. It requires many hours of experimentation and a healthy number of mistakes. But the struggle is worth it. Your code will be easier to reason about. It will be more flexible. You'll ultimately find yourself spending less time debugging, and more time creating something new and exciting.

So don't be afraid to experiment with different language features and paradigms. They're here for you to use, and they aren't going anywhere. Just remember: there's no single true paradigm, and it's never too late to throw out your old habits and learn something new.