O'REILLY®

Web Development with Node & Express

LEVERAGING THE JAVASCRIPT STACK

Ethan Brown

Web Development with Node and Express

by Ethan Brown

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Brian Anderson
Production Editor: Matthew Hacker
Copyeditor: Linley Dolby
Proofreader: Rachel Monaghan
Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

July 2014: First Edition

Revision History for the First Edition:

2014-06-27: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491949306* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Web Development with Node and Express*, the picture of a black lark and a white-winged lark, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-491-94930-6

[LSI]

Table of Contents

CHAPTER 1

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 webservers 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.