Node.js Web Development Fourth Edition

Create real-time apps with Node.js, Docker, MySQL, MongoDB, and Socket.IO. Learn about live deployment, including HTTPS and hardened security, the latest JS features and ES modules, and walk through different stages of developing robust apps using Node.js 10.

David Herron 1

Node.js is an exciting new platform for developing web applications, application servers, any sort of network server or client, and general purpose programming. It is designed for extreme scalability in networked applications through an ingenious combination of server-side JavaScript, asynchronous I/O, and asynchronous programming. It is built around JavaScript anonymous functions, and a single execution thread event-driven architecture.

While only a few years old, Node.js has quickly grown in prominence and it's now playing a significant role. Companies, both small and large, are using it for large-scale and small-scale projects. PayPal, for example, has converted many services from Java to Node.js.

The Node.js architecture departs from a typical choice made by other application platforms. Where threads are widely used to scale an application to fill the CPU, Node.js eschews threads because of their inherent complexity. It's claimed that with single-thread event-driven architectures, memory footprint is low, throughput is high, the latency profile under load is better, and the programming model is simpler. The Node.js platform is in a phase of rapid growth, and many are seeing it as a compelling alternative to the traditional web application architectures using Java, PHP, Python, or Ruby on Rails.

At its heart, it is a standalone JavaScript engine with extensions making it suitable for general purpose programming and with a clear focus on application server development. Even though we're comparing Node.js to application server platforms, it is not an application server. Instead, Node.js is a programming run-time akin to Python, Go, or Java SE. While there are web application frameworks and application servers written in Node.js, it is simply a system to execute JavaScript programs.

performance and feature improvements implemented in Chrome quickly flow through to the Node.js platform. Additionally, a team of folks are working on a Node.js implementation that runs on top of Microsoft's ChakraCore JavaScript engine (from the Edge web browser). That would give the Node.js community greater flexibility by not being reliant on one JavaScript engine provider. Visit https://github.com/nodejs/node-chakracore to take a look at the project.

The Node.js I/O library is general enough to implement any sort of server executing any TCP or UDP protocol, whether it's **domain name system** (**DNS**), HTTP, **internet relay chat** (**IRC**), or FTP. While it supports developing internet servers or clients, its biggest use case is in regular websites, in place of technology such as an Apache/PHP or Rails stack, or to complement existing websites. For example, adding real-time chat or monitoring existing websites can be easily done with the Socket. IO library for Node.js. Its lightweight, high-performance nature often sees Node.js used as a **glue** service.

A particularly intriguing combination is deploying small services using Docker into cloud hosting infrastructure. A large application can be divided into what's now called microservices that are easily deployed at scale using Docker. The result fits agile project management methods since each microservice can be easily managed by a small team that collaborates at the boundary of their individual API.

This book will give you an introduction to Node.js. We presume the following:

- You already know how to write software
- You are familiar with JavaScript
- You know something about developing web applications in other languages

We will cover the following topics in this chapter:

- Why you should use Node.js
- The architecture of Node.js
- Performance, utilization, and scalability with Node.js
- Node.js, microservice architecture, and testing
- Implementing the Twelve-Factor App model with Node.js

We will dive right into developing working applications and recognize that often the best way to learn is by rummaging around in working code.

The capabilities of Node.js

Node.js is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers! For example, there is no DOM built into Node.js, nor any other browser capability.

Beyond its native ability to execute JavaScript, the bundled modules provide capabilities of this sort:

- Command-line tools (in shell script style)
- An interactive-terminal style of program that is **Read-Eval-Print Loop** (**REPL**)
- Excellent process control functions to oversee child processes
- A buffer object to deal with binary data
- TCP or UDP sockets with comprehensive event-driven callbacks
- DNS lookup
- An HTTP, HTTPS and HTTP/2 client/server layered on top of the TCP library filesystem access
- Built-in rudimentary unit testing support through assertions

but it puts you, the programmer, very close to the protocol requests and makes you implement precisely those HTTP headers that you should return in request responses.

Typical web application developers don't need to work at a low level of the HTTP or other protocols. Instead, we tend to be more productive, working with higher-level interfaces. For example, PHP coders assume that Apache (or other HTTP servers) is already there providing the HTTP protocol, and that they don't have to implement the HTTP server portion of the stack. By contrast, a Node.js programmer does implement an HTTP server to which their application code is attached.

To simplify the situation, the Node.js community has several web application frameworks, such as Express, providing the higher-level interfaces required by typical programmers. You can quickly configure an HTTP server with baked-in capabilities such as sessions, cookies, serving static files, and logging, letting developers focus on their business logic. Other frameworks provide OAuth 2 support, or focus on REST APIs, and so on.

Node.js is not limited to web service application development. The community around Node.js has taken it in many other directions,

Build tools : Node.js has become a popular choice for developing command-line tools used in software development, or communicating with service infrastructure. Grunt and Gulp are widely used by frontend developers to build assets for websites. Babel is widely used for transpiling modern ES-2016 code to run on older browsers. Popular CSS optimizers and processors, such as PostCSS, are written in Node.js. Static website generation systems such as Metalsmith, Punch, and AkashaCMS, run at the command line and generate website content that you upload to a web server.

Web UI testing : Puppeteer gives you control over a headless-Chrome web browser instance. With it, you can develop Node.js scripts controlling a modern full-featured web browser. Typical use cases involve web scraping and testing web applications.

large chunk of Chrome, wrapped by Node.js libraries, to develop desktop applications using web UI technologies. Applications are written with modern HTML5, CSS3, and JavaScript, and can utilize leading-edge web frameworks, such as Bootstrap, React, or AngularJS. Many popular applications have been built using Electron, including the Slack desktop client application, the Atom and Microsoft Visual Code programming editors, the Postman REST client, the GitKraken GIT client, and Etcher, which makes it incredibly easy to burn OS images to flash drives to run on single-board computers.

Mobile applications : The Node.js for Mobile Systems project lets you develop smartphone or tablet computer applications using Node.js, for both iOS and Android. Apple's App Store rules preclude incorporating a JavaScript engine with JIT capabilities, meaning that normal Node.js cannot be used in an iOS application. For iOS application development, the project uses Node.js-on-ChakraCore to skirt around the App Store rules. For Android application development the project uses regular Node.js on Android. At the time of writing, the project is in an early stage of development, but it looks promising.

Internet of Things (**IoT**): Reportedly, it is a very popular language for Internet-of-Things projects, and Node.js does run on most ARM-based single-board computers. The clearest example is the NodeRED project. It offers a graphical programming environment, letting you draw programs by connecting blocks together. It features hardware-oriented input and output mechanisms, for example, to interact with **General Purpose I/O** (**GPIO**) pins on Raspberry Pi or Beaglebone single-board computers.

Server-side JavaScript

Quit scratching your head already! Of course you're doing it, scratching your head and mumbling to yourself, "What's a browser language doing on the server?" In truth, JavaScript has a long and largely unknown history outside the browser. JavaScript is a programming language, just like any other language, and the better question to ask is "Why should JavaScript remain trapped inside browsers?".

languages had just been developed. Even then, JavaScript saw use on the server side. One early web application server was Netscape's LiveWire server, which used JavaScript. Some versions of Microsoft's ASP used JScript, their version of JavaScript. A more recent server-side JavaScript project is the RingoJS application framework in the Java universe. Java 6 and Java 7 were both shipped with the Rhino JavaScript engine. In Java 8, Rhino was dropped in favor of the newer Nashorn JavaScript engine.

In other words, JavaScript outside the browser is not a new thing, even if it is uncommon.

Why should you use Node.js?

Among the many available web application development platforms, why should you choose Node.js? There are many stacks to choose from; what is it about Node.js that makes it rise above the others? We will see in the following sections.

Popularity

Node.js is quickly becoming a popular development platform with adoption by plenty of big and small players. One of those is PayPal, who are replacing their incumbent Java-based system with one written in Node.js. For PayPal's blog post about this, visit https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/ . Other large Node.js adopters include Walmart's online e-commerce platform, LinkedIn, and eBay.

According to NodeSource, Node.js usage is growing rapidly (visit https://nodesource.com/node-by-numbers). The measures include increasing bandwidth for downloading Node.js releases, increasing activity in Node.js-related GitHub projects, and more.

It's best to not just follow the crowd because the crowd claims their software platform does cool things. Node.js does some cool things, but more important is its technical merit.

JavaScript at all levels of the stack

frontend to server applications written in Java, and JavaScript was originally envisioned as a lightweight scripting language for those applets. Java never fulfilled its hype as a client-side programming language, for various reasons. We ended up with JavaScript as the principle in-browser, client-side language, rather than Java. Typically, the frontend JavaScript developers were in a different language universe than the server-side team, who was likely to be coding in PHP, Java, Ruby, or Python.

Over time, in-browser JavaScript engines became incredibly powerful, letting us write ever-more complex browser-side applications. With Node.js, we may finally be able to implement applications with the same programming language on the client and server by having JavaScript at both ends of the web, in the browser and server.

A common language for frontend and backend offers several potential benefits:

- The same programming staff can work on both ends of the wire
- Code can be migrated between server and client more easily
- Common data formats (JSON) exist between server and client
- Common software tools exist for server and client
- Common testing or quality reporting tools for server and client
- When writing web applications, view templates can be used on both sides

The JavaScript language is very popular due to its ubiquity in web browsers. It compares favorably against other languages while having many modern, advanced language concepts. Thanks to its popularity, there is a deep talent pool of experienced JavaScript programmers out there.

Leveraging Google's investment in V8

To make Chrome a popular and excellent web browser, Google invested in making V8 a super-fast JavaScript engine. Google, therefore, has a huge motivation to keep on improving V8. V8 is the JavaScript engine for Chrome, and it can also be executed standalone. Node.js is built on top of the V8 JavaScript engine.

Leaner, asynchronous, event-driven model

We'll get into this later. The Node.js architecture, a single execution thread, an ingenious event-oriented asynchronous-programming model, and a fast JavaScript engine, has less overhead than thread-based architectures.

Microservice architecture

A new sensation in software development is the microservice idea. Microservices are focused on splitting a large web application into small, tightly-focused services that can be easily developed by small teams. While they aren't exactly a new idea, they're more of a reframing of old client-server computing models, the microservice pattern fits well with agile project management techniques, and gives us more granular application deployment.

Node.js is an excellent platform for implementing microservices. We'll get into this later.

Node.js is stronger for having survived a major schism and hostile fork

During 2014 and 2015, the Node.js community faced a major split over policy, direction, and control. The **io.js** project was a hostile fork driven by a group who wanted to incorporate several features and change who's in the decision-making process. The end result was a merge of the Node.js and io.js repositories, an independent Node.js foundation to run the show, and the community is working together to move forward in a common direction.

A concrete result of healing that rift is the rapid adoption of new ECMAScript language features. The V8 engine is adopting those new features quickly to advance the state of web development. The Node.js team, in turn, is adopting those features as quickly as they show up in V8, meaning that Promises and `async` functions are quickly becoming a reality for Node.js programmers.

Threaded versus event-driven architecture

Node.js's blistering performance is said to be because of its asynchronous event-driven architecture, and its use of the V8 JavaScript engine. That's a nice thing to say, but what's the rationale for the statement?

The V8 JavaScript engine is among the fastest JavaScript implementations. As a result, Chrome is widely used not just to view website content, but to run complex applications. Examples include Gmail, the Google GSuite applications (Docs, Slides, and so on), image editors such as Pixlr, and drawing applications such as draw.io and Canva. Both Atom and Microsoft's Visual Studio Code are excellent IDE's that just happen to be implemented in Node.js and Chrome using Electron. That these applications exist and are happily used by a large number of people is testament to V8's performance. Node.js benefits from V8 performance improvements.

The normal application server model uses blocking I/O to retrieve data, and it uses threads for concurrency. Blocking I/O causes threads to wait on results. That causes a churn between threads as the application server starts and stops the threads to handle requests. Each suspended thread (typically waiting on an I/O operation to finish) consumes a full stack trace of memory, increasing memory consumption overhead. Threads add complexity to the application server as well as server overhead.

Node.js has a single execution thread with no waiting on I/O or context switching. Instead, there is an event loop looking for events and dispatching them to handler functions. The paradigm is that any operation that would block or otherwise take time to complete must use the asynchronous model. These functions are to be given an anonymous function to act as a handler callback, or else (with the advent of ES2015 promises), the function would return a Promise. The handler function, or Promise, is invoked when the operation is complete. In the meantime, control returns to the event loop, which continues dispatching events.

performance, measured in transactions per second, startup time, because that limits how quickly your service can scale up to meet demand, and memory footprint, because that determines how many application instances can be deployed per server. Node.js excels on all those measures; with every subsequent release each, is either improving or remaining fairly steady. Bailey presented figures comparing Node.js to a similar benchmark written in Spring Boot showing Node.js to perform much better. To view his talk, see https://www.youtube.com/watch?v=Fbhhc4jtGW4 .

To help us wrap our heads around why this would be, let's return to Ryan Dahl, the creator of Node.js, and the key inspiration leading him to create Node.js. In his *Cinco de NodeJS* presentation in May 2010, https://www.youtube.com/watch?v=M-sc73Y-zQA , Dahl asked us what happens while executing a line of code such as this:

```
result = query('SELECT * from db');
// operate on the result
```

Of course, the program pauses at that point while the database layer sends the query to the database, which determines the result and returns the data. Depending on the query, that pause can be quite long; well, a few milliseconds, which is an eon in computer time. This pause is bad because that execution thread can do nothing while waiting for the result to arrive. If your software is running on a single-threaded platform, the entire server would be blocked and unresponsive. If instead, your application is running on a thread-based server platform, a thread context switch is required to satisfy any other requests that arrive. The greater the number of outstanding connections to the server, the greater the number of thread context switches. Context switching is not free because more threads require more memory per thread state and more time for the CPU to spend on thread management overhead.

Simply using an asynchronous, event-driven I/O, Node.js removes most of this overhead while introducing very little of its own.

or *designing concurrent software can be complex and error prone* . The complexity comes from the access to shared variables and various strategies to avoid deadlock and competition between threads. The *synchronization primitives of Java* are an example of such a strategy, and obviously many programmers find them difficult to use. There's the tendency to create frameworks such as `java.util.concurrent` to tame the complexity of threaded concurrency, but some might argue that papering over complexity does not make things simpler.

Node.js asks us to think differently about concurrency. Callbacks fired asynchronously from an event loop are a much simpler concurrency model—simpler to understand, simpler to implement, simpler to reason about, and simpler to debug and maintain.

Ryan Dahl points to the relative access time of objects to understand the need for asynchronous I/O. Objects in memory are more quickly accessed (in the order of nanoseconds) than objects on disk or objects retrieved over the network (milliseconds or seconds). The longer access time for external objects is measured in zillions of clock cycles, which can be an eternity when your customer is sitting at their web browser ready to move on if it takes longer than two seconds to load the page.

In Node.js, the query discussed previously will read as follows:

```
query('SELECT * from db', function (err, result) {
    if (err) throw err; // handle errors
    // operate on result
});
```

The programmer supplies a function that is called (hence the name *callback function*) when the result (or error) is available. Instead of a thread context switch, this code returns almost immediately to the event loop. That event loop is free to handle other requests. The Node.js runtime keeps track of the stack context leading to this callback function, and eventually an event will fire causing this callback function to be called.

Advances in the JavaScript language are giving us new options to implement this idea. The equivalent code looks like so when used with ES2015 Promise's:

```
    // operate on result
})
.catch(err => {
    // handle errors
});
```

The following with an ES-2017 `async` function:

```
try {
     var result = await query('SELECT * from db');
     // operate on result
    } catch (err) {
     // handle errors
    }
```

All three of these code snippets perform the same query written earlier. The difference is that the query does not block the execution thread, because control passes back to the event loop. By returning almost immediately to the event loop, it is free to service other requests. Eventually, one of those events will be the response to the query shown previously, which will invoke the callback function.

With the callback or Promise approach, the `result` is not returned as the result of the function call, but is provided to a callback function that will be called later. The order of execution is not one line after another, as it is in synchronous programming languages. Instead, the order of execution is determined by the order of the callback function execution.

When using an `async` function, the coding style LOOKS like the original synchronous code example. The `result` is returned as the result of the function call, and errors are handled in a natural manner using `try/catch`. The `await` keyword integrates asynchronous results handling without blocking the execution thread. A lot is buried under the covers of the `async/await` feature, and we'll be covering this model extensively throughout the book.

where the page construction function can fire off dozens of queries—no waiting, each with their own callback—and then go back to the event loop, invoking the callbacks as each is done. Because it's in parallel, the data can be collected much more quickly than if these queries were done synchronously one at a time. Now, the reader on the web browser is happier because the page loads more quickly.

Performance and utilization

Some of the excitement over Node.js is due to its throughput (the requests per second it can serve). Comparative benchmarks of similar applications, for example, Apache, show that Node.js has tremendous performance gains.

One benchmark going around is this simple HTTP server (borrowed from https://nodejs.org/en/), which simply returns a `Hello World` message directly from memory:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

This is one of the simpler web servers that you can build with Node.js. The `http` object encapsulates the HTTP protocol, and its `http.createServer` method creates a whole web server, listening on the port specified in the `listen` method. Every request (whether a `GET` or `POST` on any URL) on that web server calls the provided function. It is very simple and lightweight. In this case, regardless of the URL, it returns a simple `text/plain` that is the `Hello World` response.

Ryan Dahl showed a simple benchmark (https://www.youtube.com/watch?v=M-sc73Y-zQA) that returned a 1-megabyte binary buffer; Node.js gave 822 req/sec, while Nginx gave 708 req/sec, for a 15% improvement over Nginx. He also noted that Nginx peaked at four megabytes memory, while Node.js peaked at 64 megabytes.

presentation was in May 2010, and Node.js has improved hugely since then, as shown in Chris Bailey's talk that we referenced earlier.

Yahoo! search engineer Fabian Frank published a performance case study of a real-world search query suggestion widget implemented with Apache/PHP and two variants of Node.js stacks (http://www.slideshare.net/FabianFrankDe/nodejs-performance-case-study). The application is a pop-up panel showing search suggestions as the user types in phrases, using a JSON-based HTTP query. The Node.js version could handle eight times the number of requests per second with the same request latency. Fabian Frank said both Node.js stacks scaled linearly until CPU usage hit 100%. In another presentation (http://www.slideshare.net/FabianFrankDe/yahoo-scale-nodejs), he discussed how Yahoo! Axis is running on Manhattan + Mojito and the value of being able to use the same language (JavaScript) and framework (YUI/YQL) on both frontend and backend.

LinkedIn did a massive overhaul of their mobile app using Node.js for the server-side to replace an old Ruby on Rails app. The switch let them move from 30 servers down to three, and allowed them to merge the frontend and backend team because everything was written in JavaScript. Before choosing Node.js, they'd evaluated Rails with Event Machine, Python with Twisted, and Node.js, choosing Node.js for the reasons that we just discussed. For a look at what LinkedIn did, see http://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedins-mobile-engineering/ .

Most existing advice on Node.js performance tips tends to have been written for older V8 versions that used the CrankShaft optimizer. The V8 team has completely dumped CrankShaft, and it has a new optimizer called TurboFan. For example, under CrankShaft, it was slower to use `try/catch` , `let/const` , generator functions, and so on. Therefore, common wisdom said to not use those features, which is depressing because we want to use the new JavaScript features because of how much it has improved the JavaScript language. Peter Marshall, an Engineer on the V8 team at Google, gave a talk at Node.js Interactive 2017 claiming that, under TurboFan, you should just write natural JavaScript. With TurboFan, the goal is for across-the-board performance improvements in V8. To view the presentation, see https://www.youtube.com/watch?v=YqOhBezMx1o .

Mikola Lysenko at Node.js Interactive 2016 went over some issues with numerical computing in JavaScript, and some possible solutions. Common numerical computing involves large numerical arrays processed by numerical algorithms that you might have learned in Calculus or Linear Algebra classes. What JavaScript lacks is multi-dimensional arrays, and access to certain CPU instructions. The solution he presented is a library to implement multi-dimensional arrays in JavaScript, along with another library full of numerical computing algorithms. To view the presentation, see https://www.youtube.com/watch?v=1ORaKEzlnys .

The bottom line is that Node.js excels at event-driven I/O throughput. Whether a Node.js program can excel at computational programs depends on your ingenuity in working around some limitations in the JavaScript language. A big problem with computational programming is that it prevents the event loop from executing and, as we will see in the next section, that can make Node.js look like a poor candidate for anything.

Is Node.js a cancerous scalability disaster?

In October 2011, software developer and blogger Ted Dziuba wrote a blog post (since pulled from his blog) titled *Node.js is a cancer* , calling it a *scalability disaster* . The example he showed for proof is a CPU-bound implementation of the Fibonacci sequence algorithm. While his argument was flawed, he raised a valid point that Node.js application developers have to consider the following: where do you put the heavy computational tasks?

A key to maintaining high throughput of Node.js applications is ensuring that events are handled quickly. Because it uses a single execution thread, if that thread is bogged down with a big calculation, Node.js cannot handle events, and event throughput will suffer.

The Fibonacci sequence, serving as a stand-in for heavy computational tasks, quickly becomes computationally expensive to calculate, especially for a naïve implementation such as this:

```
const fibonacci = exports.fibonacci = function(n) {
    if (n === 1 || n === 2) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

the best ways to calculate mathematics functions. Consider this server:

```
const http = require('http');
const url  = require('url');

const fibonacci = // as above

http.createServer(function (req, res) {
  const urlP = url.parse(req.url, true);
  let fibo;
  res.writeHead(200, {'Content-Type': 'text/plain'});
  if (urlP.query['n']) {
    fibo = fibonacci(urlP.query['n']);
    res.end('Fibonacci '+ urlP.query['n'] +'='+ fibo);
  } else {
    res.end('USAGE: http://127.0.0.1:8124?n=## where ## is the Fibonacci nu
  }
}).listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124');
```

For sufficiently large values of `n` (for example, `40`), the server becomes completely unresponsive because the event loop is not running, and instead this function is blocking event processing because it is grinding through the calculation.

Does this mean that Node.js is a flawed platform? No, it just means that the programmer must take care to identify code with long-running computations and develop solutions. These include rewriting the algorithm to work with the event loop, or rewriting the algorithm for efficiency, or integrating a native code library, or foisting computationally expensive calculations on to a backend server.

A simple rewrite dispatches the computations through the event loop, letting the server continue to handle requests on the event loop. Using callbacks and closures (anonymous functions), we're able to maintain asynchronous I/O and concurrency promises:

```
const fibonacciAsync = function(n, done) {
     if (n === 0) return 0;
    else if (n === 1 || n === 2) done(1);
```

```
const http = require('http');
const url  = require('url');

const fibonacciAsync = // as above

http.createServer(function (req, res) {
  let urlP = url.parse(req.url, true);
  res.writeHead(200, {'Content-Type': 'text/plain'});
  if (urlP.query['n']) {
    fibonacciAsync(urlP.query['n'], fibo => {
        res.end('Fibonacci '+ urlP.query['n'] +'='+ fibo);
});
  } else {
            res.end('USAGE: http://127.0.0.1:8124?n=## where ## is the Fibo
  }
}).listen(8124, '127.0.0.1'); console.log('Server running at http://127.0.0
```

Dziuba's valid point wasn't expressed well in his blog post, and it was somewhat lost in the flames following that post. Namely, that while Node.js is a great platform for I/O-bound applications, it isn't a good platform for computationally intensive ones.

Later in this book, we'll explore this example a little more deeply.

Server utilization, the business bottom line, and green web hosting

The striving for optimal efficiency (handling more requests per second) is not just about the geeky satisfaction that comes from optimization. There are real business and environmental benefits. Handling more requests per second, as Node.js servers can do, means the difference between buying lots of servers and buying only a few servers. Node.js potentially lets your organization do more with less.

Roughly speaking, the more servers you buy, the greater the cost, and the greater the environmental impact of having those servers. There's a whole field of expertise around reducing costs and the environmental impact of running web server facilities, to which that rough guideline doesn't do justice. The goal is fairly obvious—fewer servers, lower costs, and a reduced environmental impact through utilizing more efficient software.

efficiency-server-power-paper.pdf), gives an objective framework for understanding efficiency and data center costs. There are many factors, such as buildings, cooling systems, and computer system designs. Efficient building design, efficient cooling systems, and efficient computer systems (data center efficiency, data center density, and storage density) can lower costs and environmental impact. But you can destroy those gains by deploying an inefficient software stack compelling you to buy more servers than you would if you had an efficient software stack. Alternatively, you can amplify gains from data center efficiency with an efficient software stack that lets you decrease the number of servers required.

This talk about efficient software stacks isn't just for altruistic environmental purposes. This is one of those cases where being green can help your business bottom line.

Embracing advances in the JavaScript language

The last couple of years have been an exciting time for JavaScript programmers. The TC-39 committee that oversees the ECMAScript standard has added many new features, some of which are syntactic sugar, but several of which have propelled us into a whole new era of JavaScript programming. By itself, the `async/await` feature promises us a way out of what's called Callback Hell, or the situation we find ourselves in when nesting callbacks within callbacks. It's such an important feature that it should necessitate a broad rethinking of the prevailing callback-oriented paradigm in Node.js and the rest of the JavaScript ecosystem.

Refer back a few pages to this:

```
query('SELECT * from db', function (err, result) {
    if (err) throw err; // handle errors
    // operate on result
});
```

same as operations that quickly retrieve data from memory. Because of the nature of the JavaScript language, Node.js had to express this asynchronous coding construct in an unnatural way. The results do not appear at the next line of code, but instead appear within this callback function. Further, errors have to be handled in an unnatural way, inside that callback function.

The convention in Node.js is that the first parameter to a callback function is an error indicator, and the subsequent parameters are the results. This is a useful convention that you'll find all across the Node.js landscape. However, it complicates working with results and errors because both land in an inconvenient location — that callback function. The natural place for errors and results to land is on the subsequent line(s) of code.

We descend further into **callback hell** with each layer of callback function nesting. The seventh layer of callback nesting is more complex than the sixth layer of callback nesting. Why? If nothing else, it's that the special considerations for error handling become ever more complex as callbacks are nested more deeply.

```
var results = await query('SELECT * from db');
```

Instead, ES2017 async functions return us to this very natural expression of programming intent. Results and errors land in the correct location, while preserving the excellent event-driven asynchronous programming model that made Node.js great. We'll see later in the book how this works.

The TC-39 committee added many more new features to JavaScript, such as:

- A new module format that is standardized across browsers and Node.js.
- New methods for strings, such as the template string notation.
- New methods for collections and arrays — for example, operations for `map` / `reduce` / `filter`.
- The `const` keyword to define variables that cannot be changed, and the `let` keyword to define variables whose scope is limited to the block in which they're declared, rather than hoisted to the front of the function.
- New looping constructs, and an iteration protocol that works with those new loops.
- A new kind of function, the arrow function, which is lighter weight meaning less memory and execution time impact
- The Promise object represents a result that is promised to be delivered in the future. By themselves, Promises can mitigate the callback hell problem, and they form part of the basis for `async` functions.
- Generator functions are an intriguing way to represent asynchronous iteration over a set of values. More importantly, they form the other half of the basis for async functions.

You may see the new JavaScript described as ES6 or ES2017. What's the preferred name to describe the version of JavaScript that is being used?

ES1 through ES5 marked various phases of JavaScript's development. ES5 was released in 2009, and is widely implemented in modern browsers. Starting with ES6, the TC-39 committee decided to change the naming convention because of their intention to add new language features every year. Therefore, the language version name now includes the year, hence ES2015 was released in 2015, ES2016 was released in 2016, and ES2017 was released in 2017.

Deploying ES2015/2016/2017/2018 JavaScript code

fact that old browsers are still in use. Internet Explorer version 6 has fortunately been almost completely retired, but there are still plenty of old browsers installed on older computers that are still serving a valid role for their owners. Old browsers mean old JavaScript implementations, and if we want our code to work, we need it to be compatible with old browsers.

Using code rewriting tools such as Babel, some of the new features can be retrofitted to function on some of the older browsers. Frontend JavaScript programmers can adopt (some of) the new features at the cost of a more complex build toolchain, and the risk of bugs introduced by the code rewriting process. Some may wish to do that, while others will prefer to wait a while.

The Node.js world doesn't have this problem. Node.js has rapidly adopted ES2015/2016/2017 features as quickly as they were implemented in the V8 engine. With Node.js 8, we can now use async functions as a native feature, and most of the ES2015/2016 features became available with Node.js version 6. The new module format is now supported in Node.js version 10.

In other words, while frontend JavaScript programmers can argue that they must wait a couple of years before adopting ES2015/2016/2017 features, Node.js programmers have no need to wait. We can simply use the new features without needing any code rewriting tools.

Node.js, the microservice architecture, and easily testable systems

New capabilities, such as cloud deployment systems and Docker, make it possible to implement a new kind of service architecture. Docker makes it possible to define server process configuration in a repeatable container that's easy to deploy by the millions into a cloud hosting system. It lends itself best to small single-purpose service instances that can be connected together to make a complete system. Docker isn't the only tool to help simplify cloud deployments; however, its features are well attuned to modern application deployment needs.

focused, independently deployable services. They contrast this with the monolithic application deployment pattern where every aspect of the system is integrated into one bundle (such as a single WAR file for a Java EE app server). The microservice model gives developers much needed flexibility.

Some advantages of microservices are as follows:

- Each microservice can be managed by a small team
- Each team can work on its own schedule, so long as the service API compatibility is maintained
- Microservices can be deployed independently, such as for easier testing
- It's easier to switch technology stack choices

Where does Node.js fit in with this? Its design fits the microservice model like a glove:

- Node.js encourages small, tightly focused, single-purpose modules
- These modules are composed into an application by the excellent npm package management system
- Publishing modules is incredibly simple, whether via the NPM repository or a Git URL

Node.js and the Twelve-Factor app model

Throughout this book, we'll call out aspects of the Twelve-Factor App model, and ways to implement those ideas in Node.js. This model is published on http://12factor.net , and is a set of guidelines for application deployment in the modern cloud computing era. It's not that the Twelve-Factor App model is the be-all and end-all of application architecture paradigms. It's a set of useful ideas, clearly birthed after many late nights spent debugging complex applications, which offer useful ideas that could save us all a lot of effort by having easier-to-maintain and more reliable systems.

the kind of fluid self-contained cloud-deployed applications called for by our current computing environment.

Summary

You learned a lot in this chapter. Specifically, you saw that JavaScript has a life outside web browsers and you learned about the difference between asynchronous and blocking I/O. We then covered the attributes of Node.js and where it fits in the overall web application platform market and threaded versus asynchronous software. Lastly, we saw the advantages of fast event-driven asynchronous I/O, coupled with a language with great support for anonymous closures.

Our focus in this book is real-world considerations of developing and deploying Node.js applications. We'll cover as many aspects as we can of developing, refining, testing, and deploying Node.js applications.

Now that we've had this introduction to Node.js, we're ready to dive in and start using it. In Chapter 2, *Setting up Node.js* , we'll go over setting up a Node.js environment, so let's get started.

NEXT: Setting up Node.js ❯