Extracted from:

Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

The Pragmatic Bookshelf

Raleigh, North Carolina

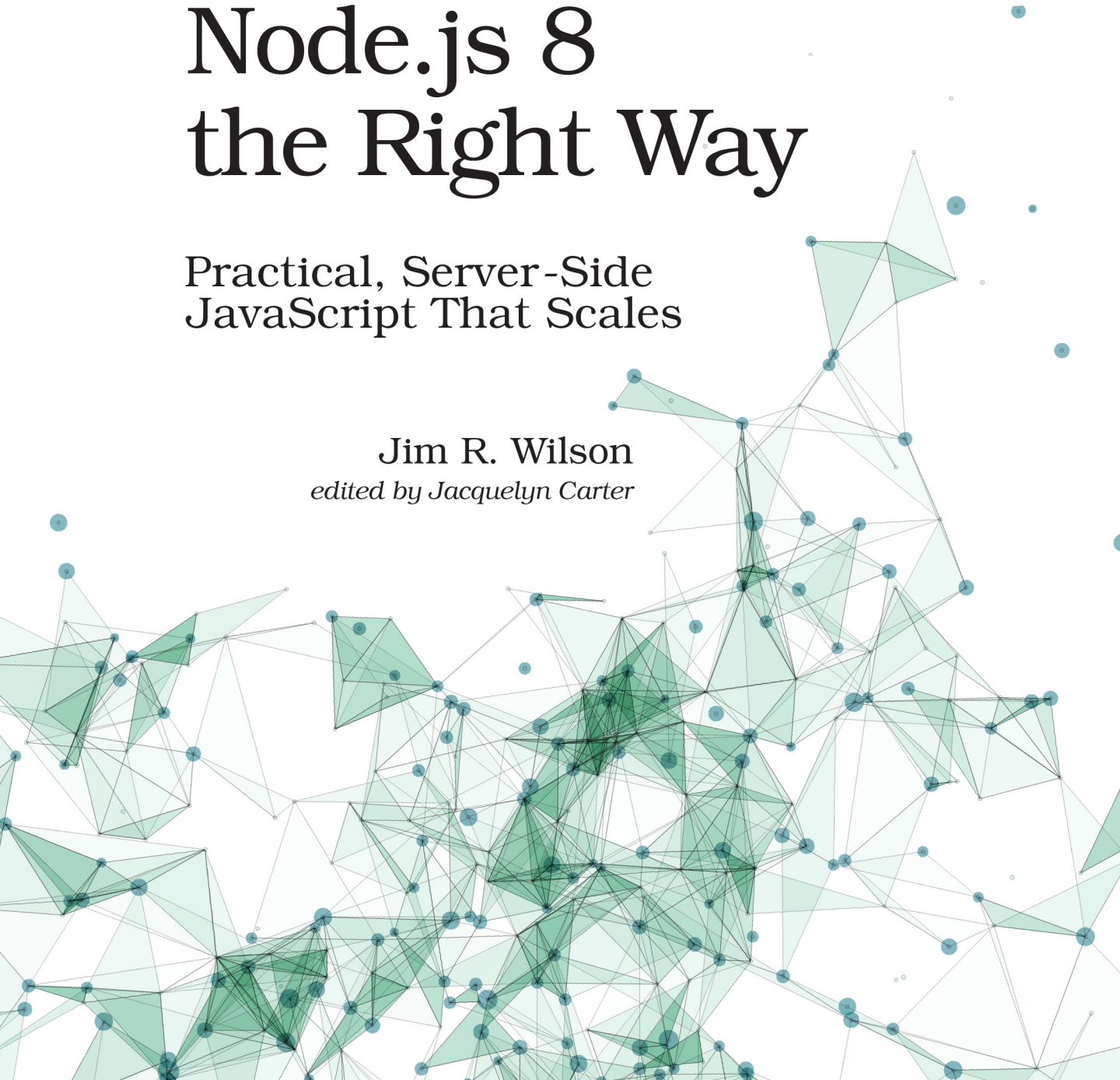
The Pragmatic Programmers

Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

edited by Jacquelyn Carter



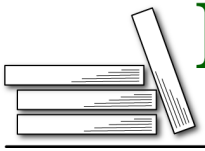
Node.js 8 the Right Way

Practical, Server-Side JavaScript That Scales

Jim R. Wilson

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Candace Cunningham
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-195-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

As a programmer, chances are you've had to access a filesystem at some point: reading, writing, renaming, and deleting files. We'll start our Node.js journey in this familiar area, creating useful, asynchronous file utilities. Along the way we'll explore the following aspects of Node.js development:

Node.js Core
: On the architecture front, you'll see how the event loop shapes a program's flow. We'll use `Buffers` for transporting data between Node.js's JavaScript engine and its native core, and we'll use Node.js's module system to bring in core libraries.

Patterns
: Inside our programs, we'll use common Node.js patterns like callbacks for handling asynchronous events. We'll harness Node.js's `EventEmitter` and `Stream` classes to pipe data around.

JavaScriptisms
: We'll take a look at some JavaScript features and best practices such as block scoping and arrow-function expressions.

Supporting Code
: You'll learn how to spawn and interact with child processes, capture their output, and detect state changes.

We'll begin by creating a tool that watches a file for changes. This'll give you a peek into how the event loop works while introducing Node.js's filesystem APIs.

Programming for the Node.js Event Loop

Let's get started by developing a couple of simple programs that watch files for changes and read arguments from the command line. Even though they're short, these applications offer insights into Node.js's event-based architecture.

Watching a File for Changes

Watching files for changes is a convenient problem to start with because it demands asynchronous coding while demonstrating important Node.js concepts. Taking action whenever a file changes is just plain useful in a number of cases, ranging from automated deployments to running unit tests.

Open a terminal to begin. Create a new directory called `filesystem` and navigate down into it.

```
$ mkdir filesystem
$ cd filesystem
```

You'll use this directory for all of the code examples in this chapter. Once there, use the touch command to create a file called target.txt.

```
$ touch target.txt
```

If you're in an environment that doesn't have the touch command (like Windows), you can alternatively echo something to write the file.

```
$ echo, > target.txt
```

This file will be the target for our watcher program. Now open your favorite text editor and enter the following:

filesystem/watcher.js

```
'use strict';
const fs = require('fs');
fs.watch('target.txt', () => console.log('File changed!'));
console.log('Now watching target.txt for changes...');
```

Save this file as watcher.js in the filesystem directory alongside the target.txt file. Although this is a short program, it deserves scrutiny since it takes advantage of a number of JavaScript and Node.js features. Let's step through it.

The program begins with the string 'use strict' at the top. This causes the program to be executed in strict mode, a feature introduced in ECMAScript version 5. Strict mode disables certain problematic JavaScript language features and makes others throw exceptions. It's always a good idea to use strict mode, and we'll use it throughout the book.

Next, notice the const keyword; this sets up fs to be a local variable with a constant value. A variable declared with const must be assigned a value when declared, and can never have anything assigned to it again (which would cause a runtime error).

It might surprise you, but it turns out that most of the time, in most code, variables don't need to be reassigned, making const a good default choice for declaring variables. The alternative to const is let, which we'll discuss shortly.

The require() function pulls in a Node.js *module* and returns it. In our case, we're calling require('fs') to incorporate Node.js's built-in filesystem module.[1]

In Node.js, a module is a self-contained bit of JavaScript that provides functionality to be used elsewhere. The output of require() is usually a plain old JavaScript object, but may also be a function. Modules can depend on other modules, much like libraries in other programming environments, which import or #include other libraries.

1. http://nodejs.org/api/fs.html

Next we call the fs module's watch() method, which takes a path to a file and a callback function to invoke whenever the file changes. In JavaScript, functions are first-class citizens. This means they can be assigned to variables and passed as parameters to other functions. Take a close look at our callback function:

```
() => console.log('File changed!')
```

This is an *arrow-function expression*, sometimes called a fat arrow function or just an arrow function. The empty pair of parentheses () at the beginning means this function expects no arguments. Then the body of the function uses console.log() to echo a message to standard output.

Arrow functions are new in ECMAScript 2015 and you'll be writing many such functions throughout this book. Prior to the introduction of arrow functions, you'd have supplied a callback using the more verbose function(){} construction:

```
function() {
  console.log('File changed!');
}
```

Aside from having a terser syntax than older function expressions, arrow functions have another big advantage over their ancestral counterparts: they do not create a new scope for this. Dealing with this has been a thorn in the side of many JavaScript developers over the years, but thanks to arrow functions, it's no longer a major source of consternation. Just like const should be your go-to means of declaring variables, arrow functions should be your first choice in declaring function expressions (such as callbacks).

The last line of the program just informs you that everything is ready. Let's try it out! Return to the command line and launch the watcher program using node, like so:

```
$ node watcher.js
Now watching target.txt for changes...
```

After the program starts, Node.js will patiently wait until the target file is changed. To trigger a change, open another terminal to the same directory and touch the file again. The terminal running watcher.js will output the string *File changed!*, and then the program will go back to waiting.

If you see duplicate messages, particularly on Mac OS X or Windows, this is not a bug in your code! There are a number of known issues around this, and many have to do with how the operating system surfaces changes.

Since you'll be touching the target file a lot this chapter to trigger changes, you might want to use the watch command to do this automatically:

```
$ watch -n 1 touch target.txt
```

This command will touch the target file once every second until you stop it. If you're on a system that doesn't have the watch command, don't worry. Any means of writing to target.txt is fine.

Visualizing the Event Loop

The program we wrote in the last section is a good example of the Node.js event loop at work. Recall the event-loop figure from *How Node.js Applications Work, on page ?*. Our simple file-watcher program causes Node.js to go through each of these steps, one by one.

To run the program, Node.js does the following:

- It loads the script, running all the way through to the last line, which produces the *Now watching* message in the console.
- It sees that there's more to do because of the call to fs.watch().
- It waits for something to happen—namely, for the fs module to observe a change to the file.
- It executes our callback function when the change is detected.
- It determines that the program still has not finished, and resumes waiting.

In Node.js the event loop spins until there's nothing left to do, there's nothing left to wait for, or the program exits by some other means. For example, if an exception is thrown and not caught, the process will exit. We'll look at how this works next.

Reading Command-Line Arguments

Now let's make our program more useful by taking in the file to watch as a command-line argument. This will introduce the process global object and how Node.js deals with exceptions.

Open your editor and enter this:

filesystem/watcher-argv.js

```
const fs = require('fs');
const filename = process.argv[2];
if (!filename) {
  throw Error('A file to watch must be specified!');
}
fs.watch(filename, () => console.log(`File ${filename} changed!`));
```

```
console.log(`Now watching ${filename} for changes...`);
```

Save the file as watcher-argv.js and run it like so:

```
$ node watcher-argv.js target.txt
Now watching target.txt for changes...
```

You should see output and behavior that's nearly identical to that of the first watcher.js program. After outputting *Now watching target.txt for changes...* the script will diligently wait for changes to the target file.

This program uses process.argv to access the incoming command-line arguments. argv stands for *argument vector*; it's an array containing node and the full path to the watcher-argv.js as its first two elements. The third element (that is, at index 2) is target.txt, the name of our target file.

Note the use of backtick characters (`) to mark the strings logged in this program:

```
`File ${filename} changed!`
```

These are called *template strings*. They can span multiple lines and they support expression interpolation, meaning you can place an expression inside of ${} and it will insert the stringified result.

If a target filename is not provided to watcher-argv.js, the program will throw an exception. You see try that by simply omitting the target.txt parameter:

```
$ node watcher-argv.js
/full/path/to/script/watcher-argv.js:4
  throw Error('A file to watch must be specified!');
  ^

Error: A file to watch must be specified!
```

Any unhandled exception thrown in Node.js will halt the process. The exception output shows the offending file and the line number and position of the exception.

Processes are important in Node. It's pretty common in Node.js development to spawn separate processes as a way of breaking up work, rather than putting everything into one big Node.js program. In the next section, you'll learn how to spawn a process in Node.

Spawning a Child Process

Let's enhance our file-watching example program even further by having it spawn a child process in response to a change. To do this, we'll bring in Node.js's child-process module and dive into some Node.js patterns and classes. You'll also learn how to use streams to pipe data around.

To keep things simple, we'll make our script invoke the ls command with the -l and -h options. This will give us some information about the target file whenever it changes. You can use the same technique to spawn other kinds of processes, as well.

Open your editor and enter this:

filesystem/watcher-spawn.js

```
'use strict';
const fs = require('fs');
const spawn = require('child_process').spawn;
const filename = process.argv[2];

if (!filename) {
  throw Error('A file to watch must be specified!');
}

fs.watch(filename, () => {
  const ls = spawn('ls', ['-l', '-h', filename]);
  ls.stdout.pipe(process.stdout);
});
console.log(`Now watching ${filename} for changes...`);
```

Save the file as watcher-spawn.js and run it with node as before:

```
$ node watcher-spawn.js target.txt
Now watching target.txt for changes...
```

If you go to a different console and touch the target file, your Node.js program will produce something like this:

```
-rw-rw-r-- 1 jimbo jimbo 6 Dec  8 05:19 target.txt
```

The username, group, and other properties of the file will be different, but the format should be the same.

Notice that we added a new require() at the beginning of the program. Calling require('child_process') returns the child process module.[2] We're interested only in the spawn() method, so we save that to a constant with the same name and ignore the rest of the module.

```
spawn = require('child_process').spawn,
```

Remember, functions are first-class citizens in JavaScript, so we're free to assign them directly to variables like we did here.

Next, take a look at the callback function we passed to fs.watch():

```
() => {
  const ls = spawn('ls', ['-l', '-h', filename]);
```

2. https://nodejs.org/api/child_process.html

```
  ls.stdout.pipe(process.stdout);
}
```

Unlike the previous arrow-function expression, this one has a multiline body; hence the opening and closing curly braces ({}).

The first parameter to spawn() is the name of the program we wish to execute; in our case it's ls. The second parameter is an array of command-line arguments. It contains the flags and the target filename.

The object returned by spawn() is a ChildProcess. Its stdin, stdout, and stderr properties are Streams that can be used to read or write data. We want to send the standard output from the child process directly to our own standard output stream. This is what the pipe() method does.

Sometimes you'll want to capture data from a stream, rather than just piping it forward. Let's see how to do that.