Node.js IN ACTION

Mike Cantelon
TJ Holowaychuk
Nathan Rajlich

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Node.js in Action version 14

For more information on this and other Manning titles go to
www.manning.com

brief contents

1 Welcome to Node.js

This chapter covers:

- What Node.js is
- JavaScript on the server
- Asyncronous and evented nature of Node
- Types of applications Node is designed for
- Sample Node programs

So what is Node.js? It's likely you have heard the term. Maybe you use Node. Maybe you are curious about it. At this point in time, Node is very popular and young (it debuted in 2009[1]). It is the second most watched project on GitHub[2], has quite a following in its Google group[3] and IRC channel[4] and has more than 15,000 community modules published in NPM (the package manager)[5]. All this to say, *there is considerable traction behind this platform*.

Footnote 1 First talk on Node by creator Ryan Dahl - http://jsconf.eu/2009/video_nodejs_by_ryan_dahl.html

Footnote 2 https://github.com/popular/starred

Footnote 3 http://groups.google.com/group/nodejs

Footnote 4 http://webchat.freenode.net/?channels=node.js

Footnote 5 http://npmjs.org

The official website (nodejs.org) defines Node as "a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js

uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

In this chapter, we'll look at:

- Why JavaScript matters for server side development
- How the browser handles I/O using JavaScript
- How Node handles I/O on the server
- What are DIRTy applications and why they are a good fit for Node
- A sampling of a few basic Node programs

Let's first turn our attention to JavaScript...

1.1 Built on JavaScript

For better or worse JavaScript is the world's most popular programming language[6]. If you have done any programming for the web, it is unavoidable. JavaScript, because of the sheer reach of the web, is the *"write once, run anywhere"* dream that Java had back in the 1990s.

Footnote 6 "JavaScript: Your New Overlord" - http://www.youtube.com/watch?v=Trurfqh_6fQ

Around the "Ajax revolution" in 2005, JavaScript went from being a "toy" language to something people write real and significant programs with. Some of the notable firsts were Google Maps and Gmail but today there are a host of web applications from Twitter to Facebook to Github.

Since the release of Google Chrome in late 2008, JavaScript performance has been improving at an incredibly fast rate due to heavy competition between browser vendors (i.e. Mozilla, Microsoft, Apple, Opera, and Google). The performance of these modern JavaScript virtual machines are literally changing the types of applications we can build on the web[7]. A compelling and frankly mind-blowing example of this is jslinux[8], a PC emulator running in JavaScript where you can load a Linux kernel, interact with the terminal session, and compile a C program all in your browser.

Footnote 7 For some examples: http://www.chromeexperiments.com/

Footnote 8 http://bellard.org/jslinux/

Node specifically uses V8, the virtual machine that powers Google Chrome, for server-side programming. V8 gives a huge boost in performance because it cuts out

the middleman, prefering straight compilation into native machine code over executing bytecode or using an interpreter. Because Node uses JavaScript on the server there are other benefits:

- Developers can write web applications in one language, which helps by: reducing the "context" switch between client and server development, and allowing for code sharing between client and server (e.g. reusing the same code for form validation or game logic).
- JSON is a very popular data interchange format today and it is native JavaScript.
- JavaScript is the language used in various NoSQL databases (e.g. CouchDB/MongoDB) so interfacing with them is a natural fit (e.g. MongoDB shell and query language is JS, CouchDB map/reduce is JS).
- JavaScript is a compilation target and there are a number of languages that compile to it already[9].

Footnote 9

https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS

- Node uses one virtual machine (V8) that keeps up with the ECMAScript [10] standard. In other words, you don’t have to wait for all the browsers to catch up to use new JavaScript language features in Node.

Footnote 10 http://en.wikipedia.org/wiki/ECMAScript

Who knew JavaScript would end up being a compelling language for writing server-side applications? Yet, due to its sheer reach, performance, and other characteristics mentioned previously, Node has gained a lot of traction. JavaScript is only one piece of puzzle though, the *way* Node uses JavaScript is even more compelling. To understand the Node environment, let's dive into the JavaScript environment we are most familiar with: the browser.

1.2 Asynchronous and Evented: The Browser

Node provides an event-driven and asynchronous platform for server-side JavaScript. It brings JavaScript to the server much in the same way a browser brings JavaScript to the client. It is important to understand how the browser works in order to understand how Node works. Both are event-driven (i.e. use an event loop[11]) and non-blocking when handling I/O (i.e. use asynchronous I/O[12]). Let's look an example to explain what that means.

Footnote 11 http://en.wikipedia.org/wiki/Event_loop

Footnote 12 http://en.wikipedia.org/wiki/Asynchronous_I/O

Take this common snippet of jQuery performing an Ajax request using XHR[13]:

Footnote 13 http://en.wikipedia.org/wiki/XMLHttpRequest

```
$.post('/resource.json', function (data) {
  console.log(data);
});
// script execution continues
```

1 I/O does not block execution

In this program we perform an HTTP request for `resource.json`. When the response comes back, an anonymous function is called (a.k.a. the "callback" in this context) containing the argument `data`, which is the data received from that request.

Notice that the code was *not* written like this:

```
var data = $.post('/resource.json');
console.log(data);
```

1 I/O blocks execution until finished

In this example, the assumption is that the response for `resource.json` would be stored in the `data` variable *when it is ready* and that the `console.log` function *will not execute until then*. The I/O operation (e.g. Ajax request) would "block" script execution from continuing until ready. Since the browser is *single-threaded*, if this request took 400ms to return, any other events happening on that page would wait until then before execution. You can imagine the poor user experience, for example, if an animation was paused or the user was trying to interact with the page somehow.

Thankfully that is not the case. When I/O happens in the browser, it happens outside of the event loop (i.e. main script execution) and then an "event" is emitted when the I/O is finished[14] which is handled by a function (often called the "callback") as seen in figure 1.1:

Footnote 14 For completeness, there are a few exceptions which "block" execution in the browser and usage is typically discouraged: alert, prompt, confirm, and synchronous XHR.

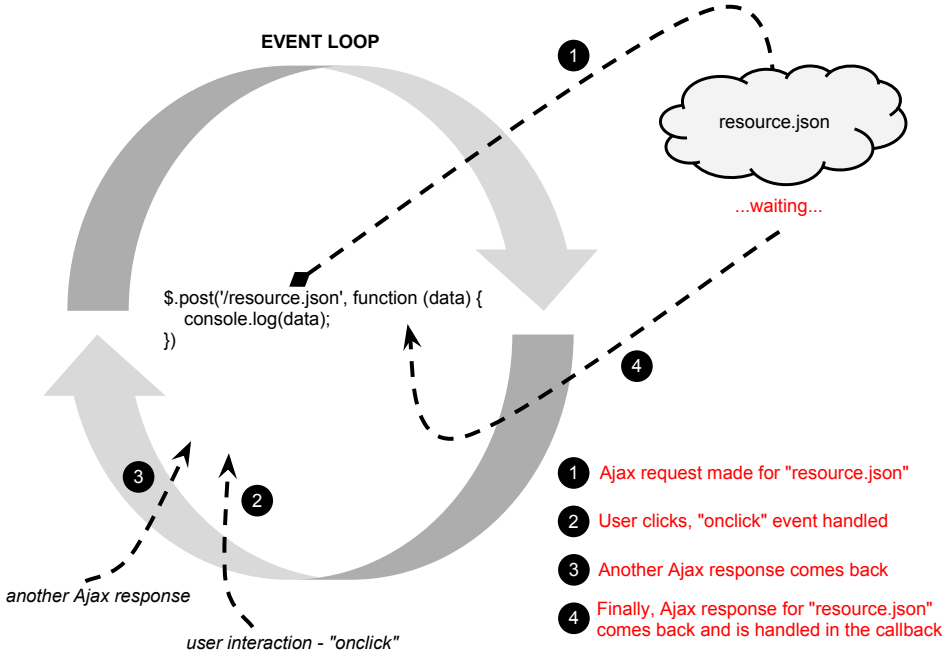

Figure 1.1 An example of non-blocking I/O in the browser

The I/O is happening asynchronously and not "blocking" the script execution allowing the event loop to respond to whatever other interactions or requests that are being performed on the page. This enables the browser to be responsive to the client and handle a lot of interactivity on the page.

Make a note of that and let's switch over to the server.

1.3 Asynchronous and Evented: The Server

For the most part, we are familiar with a conventional I/O model for server-side programming like the "blocking" jQuery example in section 1.2. Here's an example of how it looks in PHP:

```
$result = mysql_query('SELECT * FROM myTable');
print_r($result);
```

1 Execution stops until DB query completes

We are doing some I/O and the process is blocked from continuing until all the data has come back. For many applications this model is fine and is easy to follow. What may not be apparent is that the process has state/memory and is essentially doing "nothing" until the I/O is completed. That could take 10ms to minutes depending on the latency of the I/O operation. Latency can be unexpected as well, for example:

- The disk is performing a maintainence operation, pausing reads/writes
- A database query is slower because of increased load
- Pulling a resource from "sitexyz.com" is sluggish today for some reason

If a program blocks on I/O, what does the server do when there are more requests to handle? Typically this means that we use a multi-threaded approach. A common implementation is to use one thread per connection and setup a thread pool for those connections. You can think of threads as computational workspaces in which the processor works on one task. In many cases, a thread is contained inside a process and maintains its own working memory. Each thread handles one or more server connections. While this sounds like a natural way to delegate server labor, to developers who've been doing this a long time, managing threads within an application can be complex. Also, when a large number of threads is needed to handle many concurrent server connections, threading can tax operating system resources. Threads require CPU to perform context-switches as well as additional RAM.

To illustrate this, let's look at a benchmark (shown in Figure 1.2) comparing NGINX and Apache. NGINX[15], if you aren't familiar with it, is an HTTP server like Apache but instead of using the multi-threaded approach with blocking I/O, it uses an event loop with asynchronous I/O (like the browser and Node). Because of these design choices, NGINX is often able to handle more requests and connected clients, making it a more responsive solution.[16]

Footnote 15 http://nginx.com/

Footnote 16 If you are more interested in this problem: http://www.kegel.com/c10k.html

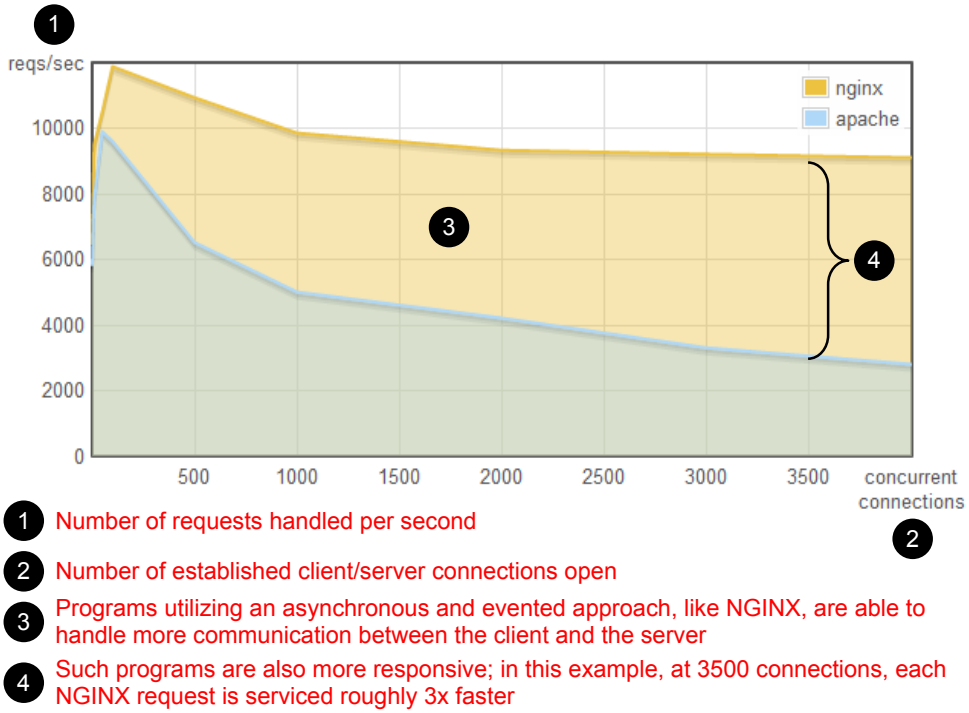

Figure 1.2 WebFaction Apache / NGINX benchmark: http://jppommet.com/from-webfaction-a-little-holiday-present-1000

In Node, I/O almost always is performed outside of the main event loop allowing the server to stay efficient and responsive like NGINX. This makes it much harder for a process to become I/O bound because I/O latency isn't going to crash your server or use the resources it would if you were blocking. It allows the server to be lightweight on what are typically the slowest operations a server performs.[17].

Footnote 17 More details at http://nodejs.org/about/

This mix of an event-driven and asynchronous model and the widely accessible JavaScript language helps open up a exciting world of data-intensive real-time applications.

1.4 DIRTy Applications

There actually is acronym for the types of applications Node is designed for: *DIRT*. It stands for *data-intensive real-time* applications. Since Node itself is very lightweight on I/O, it is good at shuffling/proxying data from one pipe to another. It allows a server to hold open a number of connections while handling many requests and keeping a small memory footprint. It is designed to be responsive like the browser.

Real-time applications are a new use-case of the web. Many web applications

now provide information virtually instantly, implementing things like: online whiteboard collaboration, realtime pinpointing of approaching public transit buses, and multiplayer games. Whether its existing applications being enhanced with real-time components or completely new types of applications, the web is moving towards more responsive and collaborative environments. These new types of web applications, however, call for a platform that can respond almost instantly to a large number of concurrent users. Node is good at this and not just for web, but also other I/O heavy applications.

A good example of a DIRTy application written with Node is Browserling[18] (shown in Figure 1.3). The site allows in-browser use of other browsers. This is extremely useful to front-end web developers as it frees them from having to install numerous browsers and operating systems solely for testing. Browserling leverages a Node-driven project called StackVM, which manages virtual machines (VMs), created using the QEMU ("Quick Emulator") emulator. QEMU emulates the CPU and peripherals needed to run the browser.

Footnote 18 browserling.com

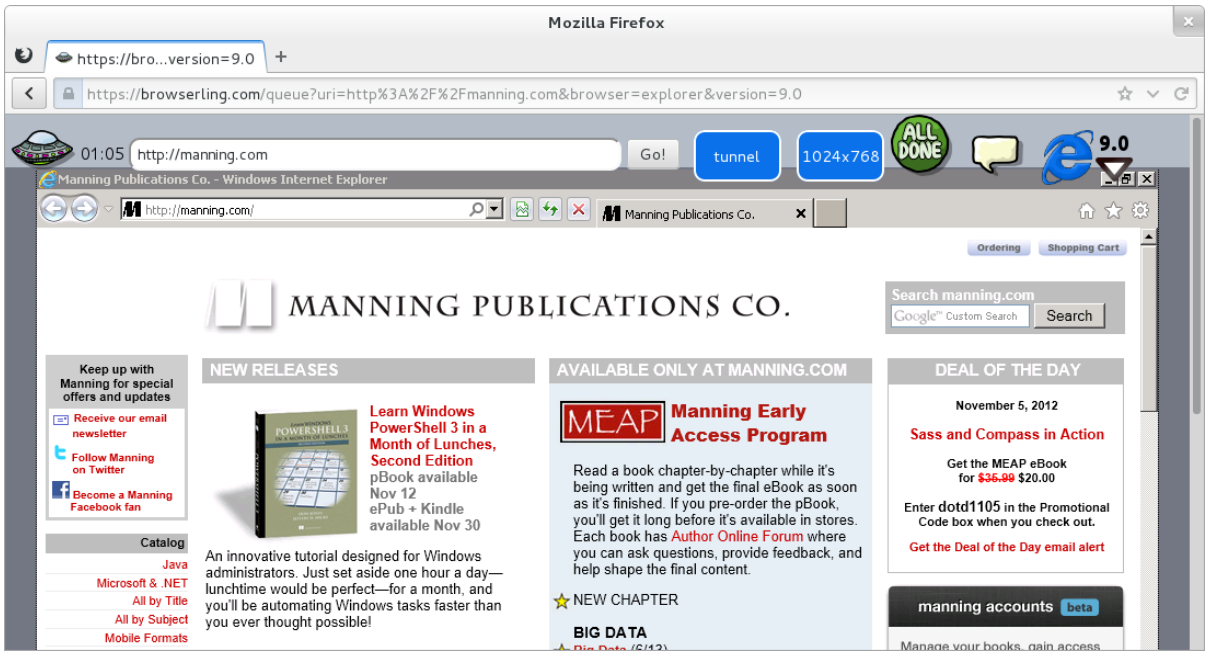

Figure 1.3 Browserling: Interactive cross-browser testing utilizing Node.js

Browserling has VMs run test browsers and then relays the keyboard and mouse input data from the user's browser to the emulated browser which, in turn, streams the repainted regions of the emulated browser and redraws them on the canvas of the user's browser. This is illustrated in figure 1.4.

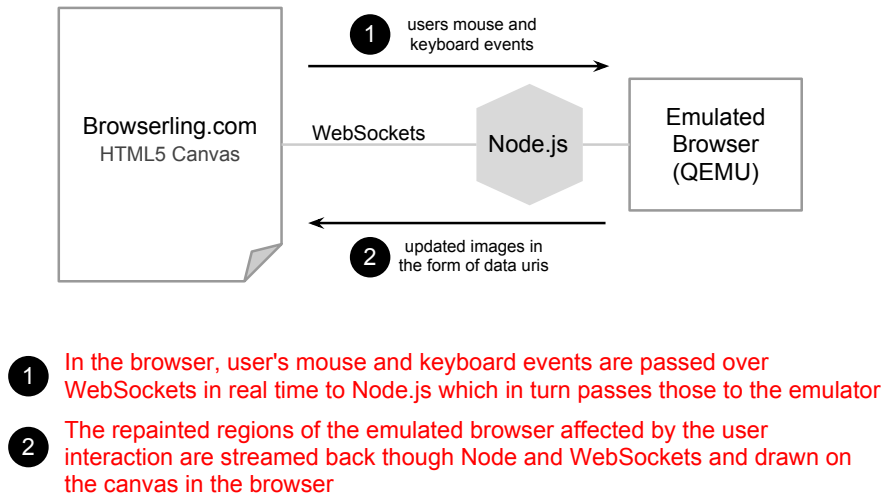

Figure 1.4 Browserling Workflow

Browserling also provides a complimentary project using Node called Testling [19] allowing you to run a tests suites against multiple browsers in parallel from the command line.

Footnote 19 testling.com

Browserling and Testling are good examples of a DIRTy applications and the infrustructure for building a scalable network applications like it are at play when you sit down to write your first Node application. Let's take a look at how Node's API provides this tooling right out of the box.

1.5 DIRTy by Default

Node was built from the ground up to have an event-driven and asynchronous model. JavaScript has never had standard I/O libraries, which are common to server-side languages, the "host" environment has always determined this for JavaScript. The most common "host" environment for JavaScript, the one most developers are used to, is the browser which is event-driven and asynchronous.

Node tries to keep consistency between the browser by reimplementing common "host" objects, for example:

- Timer API (e.g. `setTimeout`)

- Console API (e.g. `console.log`)

Node also includes a "core" set of modules for many types of network and file I/O. These include modules for: HTTP, TLS, HTTPS, File System (POSIX), Datagram (UDP), and NET (TCP). The "core" is intentially small, low-level, and uncomplicated including just the building blocks for I/O based applications. 3rd-party modules build upon these blocks to offer greater abstractions for common problems.

SIDEBAR **Platform vs. Framework**

Node is a platform for JavaScript applications which is not to be confused with a framework. It is a common misconception to think of Node as "Rails" or "Django" for JavaScript when it is much lower level. Although if you are interested in frameworks for web applications, we will talk about a popular one for Node called Express later on in this book.

After all this discussion you probably are wondering what does Node code look like? Let's cover a few simple examples:

- A simple asyncronous example
- A "hello world" web server
- An example of streams

1.5.1 Simple Async Example

In section 1.2, we looked at this Ajax example using jQuery:

```
$.post('/resource.json', function (data) {
  console.log(data);
});
```

Let's do something similar in Node but instead using the file system (`fs`) module to load `resource.json` from disk. Notice how similar the program is compared to the previous jQuery example.

```
var fs = require('fs');
fs.readFile('./resource.json', function (er, data) {
  console.log(data);
```

```
})
```

In this program, we read the `resource.json` file from disk. When all the data is read, an anonymous function is called (a.k.a. the "callback") containing the arguments `er`, if any error occured, and `data`, which is the file data.

The process "loops" behind the scenes able to handle any other operations that may come its way until the data is ready. All the evented/async benefits we talked about earlier are in play automatically. The difference here is that instead of making an Ajax request from the browser using jQuery, we are accessing the file system in Node to grab `resource.json`. This latter action is illustrated in figure 1.5.

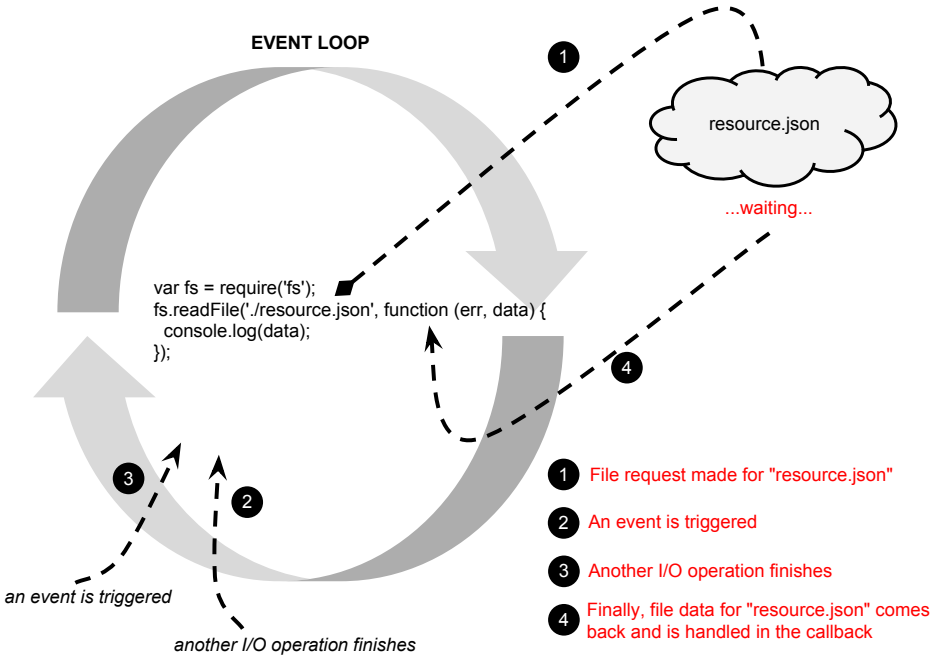

Figure 1.5 An example of non-blocking I/O in Node

1.5.2 Hello World HTTP Server

A very common use case for Node is building servers. Node makes it very simple to create different types of servers. This can feel odd if you are used to having a server 'host' your application (e.g. a PHP application hosted on Apache HTTP server). In Node, the server and the application *are the same*. Here is an example of an HTTP server that simply responds to any request with 'Hello World':

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

Whenever a request happens, the `function (req, res)` "callback" is fired and "Hello World" is written out as the response. This event model is akin to listening to an `onclick` event in the browser. A 'click' could happen at any point so you setup a function to perform some logic whenever that happens. Here, Node provides a function for a 'request' whenever that happens. Another way to write this same server to make the 'request' event even more explicit is:

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
})
server.listen(3000);
console.log('Server running at http://localhost:3000/');
```

1 Setting up an event listener for 'request'

1.5.3 Streaming Data

Node is also huge on streams and streaming. You can think of streams like arrays but instead of having data distributed over space, streams can be thought of as *data distributed over time*. By bringing data in chunk by chunk, the developer is given the ability to handle that data as it comes in instead of waiting for it all to arrive before acting. Here's how we would stream `resource.json`:

```
var stream = fs.createReadStream('./resource.json')
stream.on('data', function (chunk) {
  console.log(chunk)
})
stream.on('end', function () {
  console.log('finished')
})
```

1 'data' event fires when new chunk is ready

`data` events are fired whenever a new chunk of data is ready and `end` is fired

when all the chunks have been loaded. A 'chunk' can vary in size depending on the type of data. This low level access to the read stream allows you efficiently deal with data as it is read instead of waiting for it all to buffer in memory.

Node also provides writable streams that you can write chunks of data to. One of those is the response (`res`) object when a 'request' happens on an HTTP server.

Readable and writeable streams can be connected to make pipes much like the `| (pipe)` operator in shell scripting. This provides an efficient way to write out data as soon as it's ready without waiting for the complete resource to be read and then written out. Let's use our HTTP server from before to illustrate streaming an image to a client:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'image/png'});
  fs.createReadStream('./image.png').pipe(res);
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

1 Piping from a readable stream to a writeable stream.

In this one-liner, the data is read in from the file (`fs.createReadStream`) and being sent out (`.pipe`) to the client (`res`) as it comes in. The event loop is able to handle other events while data is being streamed.

Node provides this "DIRTy by default" approach across multiple platforms including various UNIXes and Windows. The underlying asynchronous I/O library (libuv) was built specifically to provide a unified experience regardless of the parent operating system which allows programs to be more easily ported across devices and run on multiple devices if needed.

1.6 In Summary

Like any technology, Node is not a silver bullet, it just tries to tackle certain problems and open new possibilities. One of the interesting things about Node is it brings people from all aspects of the system together. Many come to Node being JavaScript client-side programmers, others are server-side programmers, and others are systems level programmers. Wherever you fit, we hope you have an understanding of where Node may fit in your stack. To review:

Node is:

- Built on JavaScript

- Evented and asynchronous
- For data-intensive real-time applications

In Chapter 2, we will build a simple DIRTy web application so you can see how a Node application works.