



**INSTANT**

Short | Fast | Focused

# Node.js Starter

Program your scalable network applications and web services  
with Node.js

Pedro Teixeira

**[PACKT]**  
PUBLISHING

# Instant Node.js Starter

Program your scalable network applications and web services with Node.js

**Pedro Teixeira**



BIRMINGHAM - MUMBAI

# Instant Node.js Starter

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2013

Production Reference: 1170513

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-78216-556-9

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Author**

Pedro Teixeira

**Project Coordinator**

Esha Thakker

**Reviewer**

Andris Reinman

**Proofreader**

Paul Hindle

**Acquisition Editor**

Akram Hussain

**Production Coordinator**

Shantanu Zagade

**Commissioning Editor**

Sharwari Tawde

**Cover Work**

Shantanu Zagade

**Technical Editor**

Saijul Shah

**Cover Image**

Nilesh R. Mohite

**Copy Editor**

Brandt D'Mello

# About the Author

**Pedro Teixeira** is a prolific open source programmer and author of many Node.js modules. After he graduated as a software engineer over 14 years ago, he has been a consultant, a programmer, and an active and internationally known Node.js community member.

He is a founding partner of The Node Firm (a consulting company specializing in Node.js) and author of the popular Node Tuts screencasts ([www.nodetuts.com](http://www.nodetuts.com)) and three other books about Node.js, namely, *Hands-on Node.js* (self-published), *Professional Node.js*, *Wrox*, and *Node.js for UI Testing*, Packt Publishing.

When he was ten years old, his father taught him how to program a ZX Spectrum, and since then, he never wanted to stop. He taught himself how to program his father's Apple IIc and then entered the PC era. During college, he was introduced to the universe of Unix and open source, becoming seriously addicted to it. In his professional life, he has developed systems and products with Visual Basic, C, C++, Java, PHP, Ruby, and JavaScript for big Telco companies, banks, hotel chains, and others.

He has been a Node.js enthusiast since the very beginning, having authored many applications and also many well-known modules, such as Fugue, Alfred.js, Carrier, Nock, and others.

He's the organizer of LXJS— the Lisbon JavaScript Conference.

---

I would like to thank my amazing wife, Susana, for her support and resilience; you have always been a cornerstone for me.

I'd also like to thank the amazing JavaScript and Node.js community for being so enthusiastic and innovative, always taking everyone along on crazy rides, and being at the fulcrum of expanding the reach and capabilities of programmers.

---

# About the Reviewer

**Andris Reinman** is a Node.js/JavaScript developer from Tallinn, Estonia. Even though he holds a degree in Electrical Engineering, Andris somehow wandered off to the magical land of JavaScript and has been hooked ever since. He started using JavaScript with the Ajax revolution back in 2005, and turned into a full-time JavaScript developer two years later. For the past three years, Andris has been developing mostly Node.js-based programs. Andris is known for his open source e-mail modules for Node.js with the SMTP client Nodemailer, which is the first-choice e-mail-sending library for any Node.js developer.

Andris works for Pipedrive Inc. and has helped in creating one of the best sales pipeline management tools in the world ([www.pipedrive.com](http://www.pipedrive.com)). Previously, he worked for NETI.ee, a local search engine and website directory in Estonia.

# www.packtpub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.packtpub.com](http://www.packtpub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.packtpub.com](http://www.packtpub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

# packtlib.packtpub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- ◆ Fully searchable across every book published by Packt
- ◆ Copy and paste, print and bookmark content
- ◆ On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.packtpub.com](http://www.packtpub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.







# Table of Contents

<b>Instant Node.js Starter</b>	<b>1</b>
So, what is Node.js?	3
Installation	4
Quick start	9
Step 1 – Creating a Hello World HTTP server	9
Step 2 – Launching the server	9
<b>Top 5 features you need to know about</b>	<b>12</b>
User modules	12
Node Package Manager (NPM)	14
The callback pattern	17
Chaining I/O	19
Event emitter	20
Parallelizing I/O	22
Streams	23
The readable streams	24
The writable streams	25
The duplex streams	25
Stream flow control	27
Piping	28
The transform streams	29
<b>People and places you should get to know</b>	<b>31</b>
Official sites	31
Blogs	31
API documentation	31
NPM modules	31
Github repository	31
Community	31
Twitter	32
About Packt Publishing	35
Writing for Packt	35



# Instant Node.js Starter

Welcome to *Instant Node.js Starter*. This book has been especially created to provide you with all the information that you need to get up to speed with Node.js, a platform for building fast and scalable network applications. You will learn the basics of nodes, get started with building your first Node.js HTTP server, and learn about the five main building blocks of Node.js: modules, callback functions, the event emitter, streams, and NPM.

This book contains the following sections:

*So what is Node.js?* helps you find out what Node.js actually is, what you can do with it, and why it's so great.

*Installation* teaches you how to download and install Node.js with the minimum fuss and then set it up so that you can use it as soon as possible.

*Quick start* will show you how to start building an HTTP server using Node.js.

*Top 5 features you need to know about* will teach you how to perform five tasks involving the most important features of Node.js. By the end of this section, you will understand and be able to create modules, install third-party modules via NPM, and use the three main patterns of Node.js: callback functions, event emitters, and streams.

*People and places you should get to know* provides you with many useful links to the project page and forums as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of Node.js super-contributors. This is vital, because every open source project is centered around a community.



## So, what is Node.js?

Node.js is an open source platform that allows you to build fast and scalable network applications using JavaScript. Node.js is built on top of V8, a modern JavaScript virtual machine that powers Google's Chrome web browser.

Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js can handle multiple concurrent network connections with little overhead, making it ideal for data-intensive, real-time applications.

With Node.js, you can build many kinds of networked applications. For instance, you can use it to build a web application service, an HTTP proxy, a DNS server, an SMTP server, an IRC server, and basically any kind of process that is network intensive.

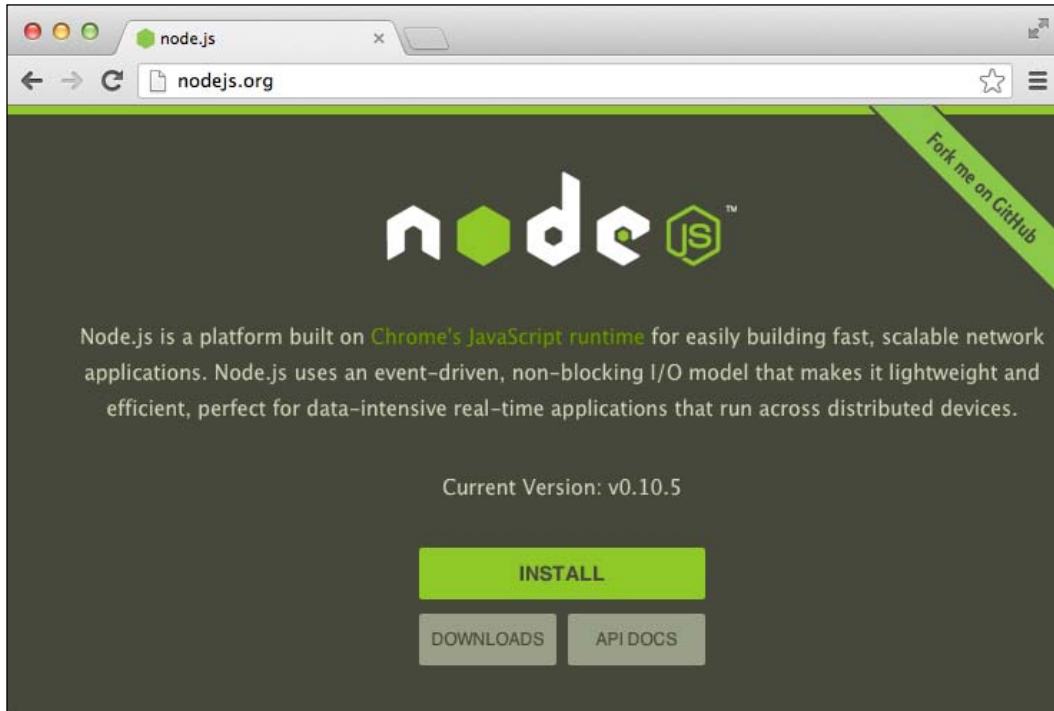
You program Node.js using JavaScript, which is the language that powers the Web. JavaScript is a powerful language that, when mastered, makes writing networked, event-driven applications fun and easy.

Node.js recognizes streams that are resistant to precarious network conditions and misbehaving clients. For instance, mobile clients are notoriously famous for having large latency network connections, which can put a big burden on servers by keeping around lots of connections and outstanding requests. By using streaming to handle data, you can use Node.js to control incoming and outgoing streams and enable your service to survive.

Also, Node.js makes it easy for you to use third-party open source modules. By using **Node Package Manager (NPM)**, you can easily install, manage, and use any of the several modules contained in a big and growing repository. NPM also allows you to manage the modules your application depends on in an isolated way, allowing different applications installed in the same machine to depend on different versions of the same module without originating a conflict, for instance. Given the way it's designed, NPM even allows different versions of the same module to coexist in the same application.

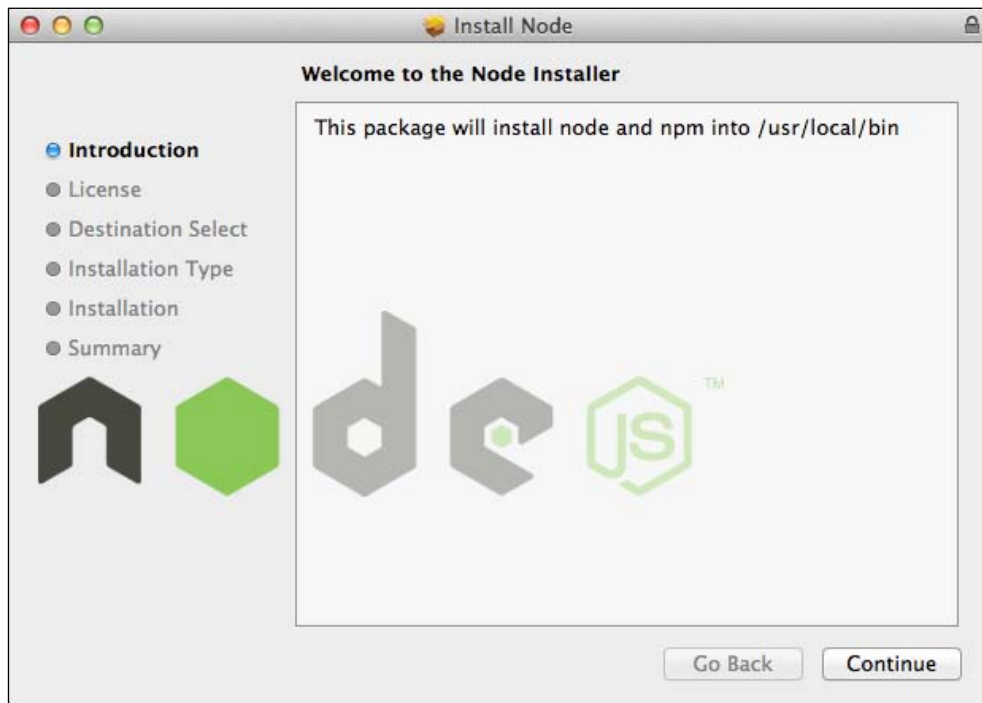
## Installation

To install Node.js on a Windows or Macintosh machine, go to the <http://Nodejs.org> website and click on the **INSTALL** button.

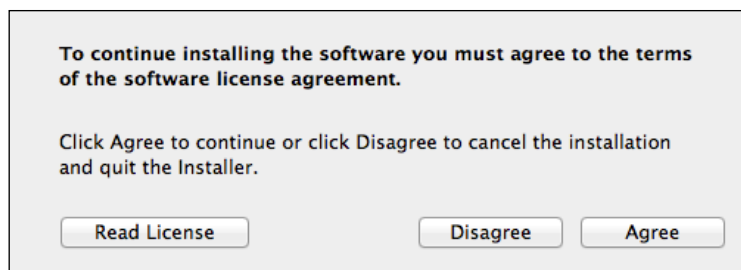


The browser will then ask you if you wish to download a file coming from the [Nodejs.org](http://Nodejs.org) website, to which you should agree.

Once the file is downloaded, execute it, launching a graphical installer as shown in the following screenshot:

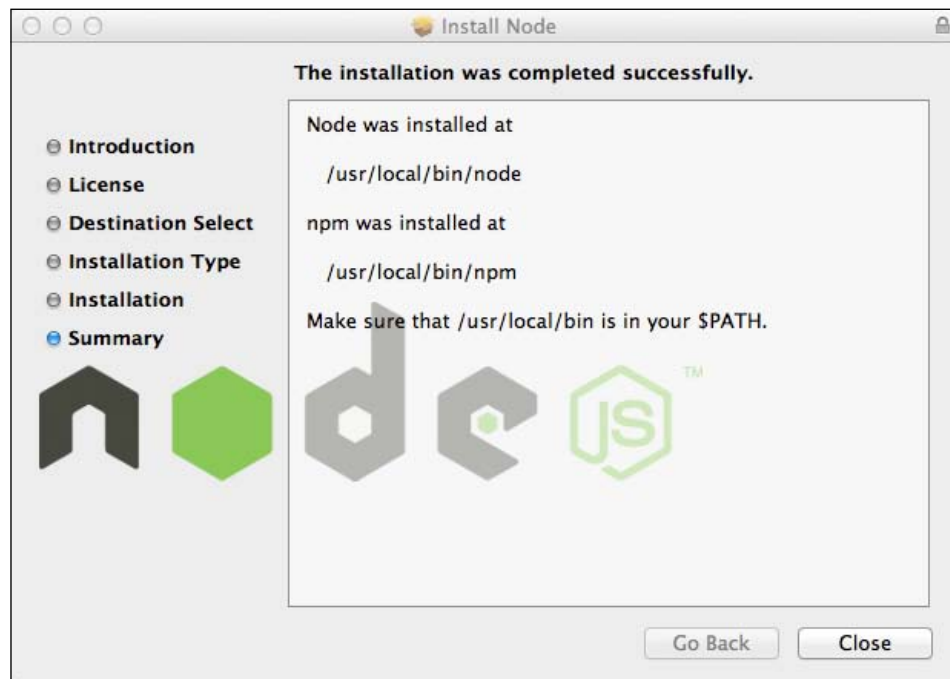


When you click on **Continue**, the installer will prompt you to accept the Software License agreement.





If you agree with the terms, click on the **Agree** button (if you run a Mac OS) or check an agreement checkbox and click on the **Next** button (if you run a Windows OS); then click on the **Install** button that follows it. On some systems, the installer will prompt you for a system username and password that gives you permission to Install Node globally.

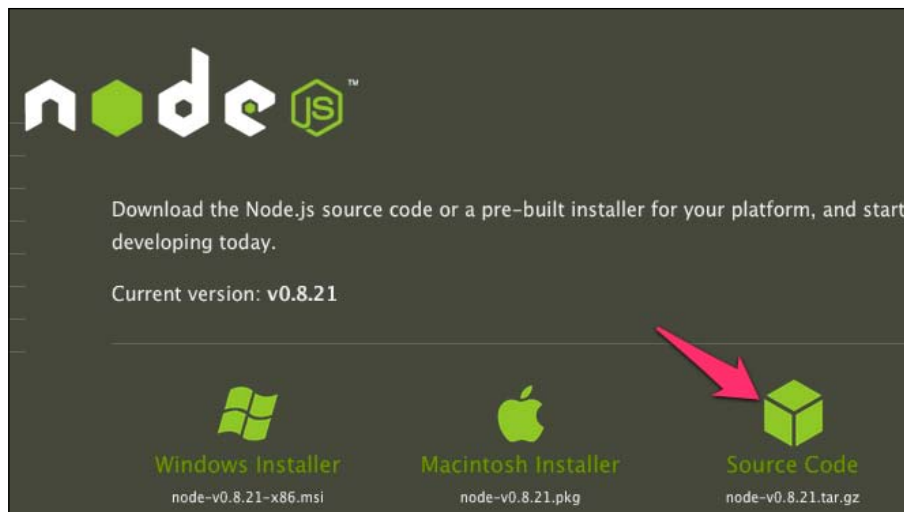


After Node is installed in your system, the installer will show you a success screen. The Mac OS installer will state where both the Node and the NPM executables were installed.

If you don't have a Windows or Macintosh machine, Node also installs on the various Unix and Linux OS distributions using the appropriate package manager. The Node.js project maintains a wiki page with the installation guide for some distributions on the following page:

<https://github.com/joyent/Node/wiki/Installing-Node.js-via-package-manager>

Some distributions may not keep the latest stable Node.js version available. If that's the case or if you simply don't have another option, you can always download, build, and install Node.js from the source code. For that, head out to the download page of the `Nodejs.org` website (<http://Nodejs.org/download/>) and click on the **Source Code** icon. That will start the download of the source code.



Once the source code tarball is downloaded, go to this wiki page:  
<https://github.com/joyent/Node/wiki/Installation>.

This is a wiki page maintained by the Node.js project that explains what further steps you need to take to compile and install Node from the source code.

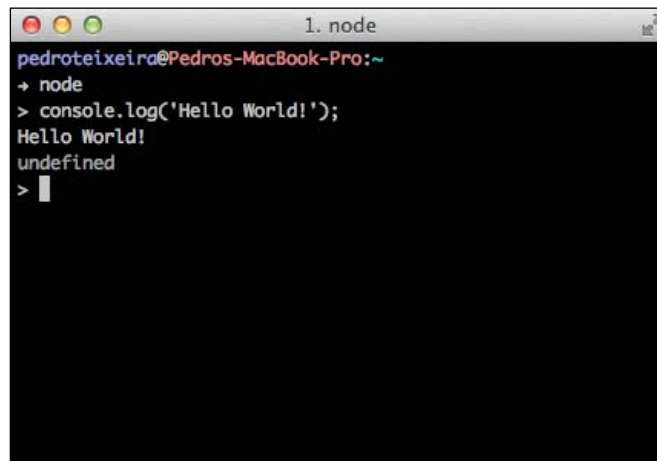
Once you have Node.js installed on your machine, you can certify that you have it available by opening a shell prompt and typing `node` followed by `Enter`. That should open the Node.js **Read-Eval-Print Loop (REPL)**, which is a Node interactive console to which you can send JavaScript commands.

```
1. node
Last login: Tue Mar  5 14:09:06 on ttys000
pedroteixeira@Pedros-MacBook-Pro:~
→ node
> 
```

You can then try and write something to the console by typing the following command:

```
console.log('Hello World!');
```

Once you hit *Enter*, you will see the **Hello World!** string written out to the console. You will also see **undefined** written below that. This is because the REPL will print the value of the expression you enter there. In this case, this means that the `console.log` function call returns `undefined`.

A screenshot of a terminal window titled "1. node". The prompt is "pedroteixeira@Pedros-MacBook-Pro:~". The user enters "node", followed by "> console.log('Hello World!');". The output shows "Hello World!" on one line and "undefined" on the next line. The prompt "> " is followed by a cursor.

```
1. node
pedroteixeira@Pedros-MacBook-Pro:~
→ node
> console.log('Hello World!');
Hello World!
undefined
> 
```

## Quick start

Now that we have a working Node installation in place, we will see how easy it is to create and launch a new server with just a few keystrokes.

### Step 1 – Creating a Hello World HTTP server

Let's then create our first Hello World server in Node.js. For that, create a file named `hello_world_http_server.js` with the following content:

```
var http = require('http');
var server = http.createServer();
server.on('request', function(req, res) {
  res.writeHead(200, {'content-type': 'text/plain'});
  res.write('Hello World!');
  res.end();
});

var port = 8080;
server.listen(port);
server.once('listening', function() {
  console.log('Hello World server listening on port %d', port);
});
```



#### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

### Step 2 – Launching the server

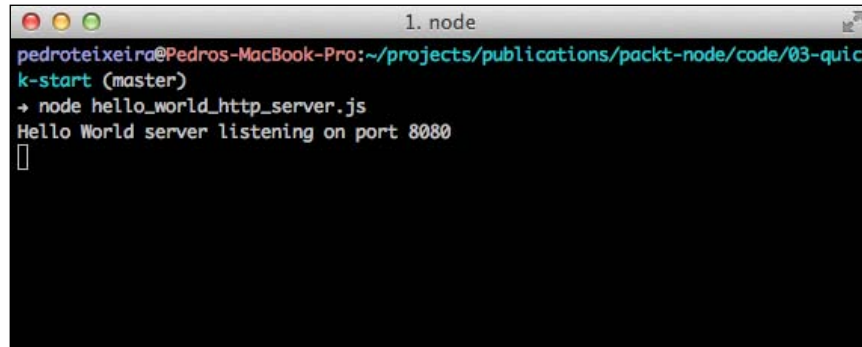
Once you have this file saved, launch it from the command-line prompt by using the Node executable, like this:

```
$ Node hello_world_http_server.js
```

## Instant Node.js Starter

---

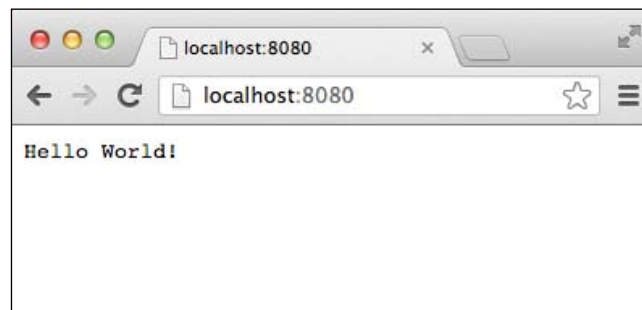
Once you press *Enter*, it should immediately output this:

A terminal window titled "1. node" on a Mac. The prompt is "pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/03-quick-start (master)". The user enters the command "node hello\_world\_http\_server.js". The output is "Hello World server listening on port 8080" followed by a cursor.

```
1. node
pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/03-quick-start (master)
→ node hello_world_http_server.js
Hello World server listening on port 8080
█
```

Now you can connect to it using a web browser, pointing the browser to `http://localhost:8080`.

You should then see the **Hello World!** string rendered in the browser as shown in the following screenshot. Congratulations! You have successfully connected to your first Node.js HTTP server! Now that you've done that, we can explain to you step by step the code that you just created.



The first line is:

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

Here, you are requiring the Node core `http` module. The Node core API is decomposed into several modules, and here we are looking up the module named `http`, specifying this name inside the `require` function call. The call to `require` returns the HTTP module value, which we then assign to a variable conveniently named `http`. This variable could have been named whatever we wanted to name it; it simply serves the purpose of having a reference to the Node HTTP module for the code that follows.

Then we use this module to create an HTTP server in the following line:

```
var server = http.createServer();
```

This line creates an HTTP server that we'll later wire up to make it behave as we want.

Then we have the following block of code:

```
server.on('request', function(req, res) {  
  res.writeHead(200, {'content-type': 'text/plain'});  
  res.write('Hello World!');  
  res.end();  
});
```

Here we're binding a function to the server `request` event. This event gets fired once the server receives a new HTTP request. Since you have this function bound to that event, it gets called for every request that this server answers, having two objects passed in as arguments: an HTTP server request and an HTTP server response. You can then use this last object to reply to the client.

The first thing we do when we get a request is write out an HTTP header specifying the response content type. Here we're setting the content type to `text/plain`, as an example:

```
res.writeHead(200, {'content-type': 'text/plain'});
```

Next, we're using the HTTP server response object to write out a string to the browser:

```
res.write('Hello World!');
```

After that, we end the response, which is required for the HTTP protocol to end and the browser to know that the response has ended:

```
res.end();
```

After we are bound to the `request` event, we make the HTTP server listen to a specific TCP port:

```
var port = 8080;  
server.listen(port);
```

This makes the server available for requests on the TCP port 8080. Once the server is listening on that port, the server emits a `listening` event. By listening to that event, we can print out a message once the server is available:

```
server.once('listening', function() {  
  console.log('Hello World server listening on port %d', port);  
});
```

Here, we're using a variant of the `server.on` method, named `server.once`, which behaves in the same way but only cares about the first time that the event occurs. Since, in our case, the `listening` event will be fired only once, using `server.once` is more appropriate than `server.on`.

## Top 5 features you need to know about

In the last section, we included the `http` module in the local application by using the `require` function, passing it a name of the module you wish to include. This function can be used for including and using any other module that `Node.js` comes with. Here is a short list of some of the core modules:

- ◆ `net`: For creating TCP clients and servers
- ◆ `http`: For creating and consuming HTTP services
- ◆ `fs`: For accessing and manipulating files
- ◆ `dns`: For using the DNS service
- ◆ `events`: For creating event emitters
- ◆ `stream`: For creating streams
- ◆ `os`: For accessing some local operating system statistics
- ◆ `assert`: For assertion testing
- ◆ `util`: For miscellaneous utilities

You can, for instance, import the `fs` module into a local variable by running the following module in your JavaScript code:

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

## User modules

Node.js supports three main types of modules: core modules, user modules, and third-party modules. Let's now see how you can use the Node.js module system to create and use modules of your own.

In Node.js, a module is simply a file. To test this, let's create the simplest of modules in a file named `simple_module.js`:

```
module.exports = 'ABC';
```

This is a module that simply exports the string `ABC`.

Then, in the same directory, create another module named `import.js` that will use this file:

```
var simpleModule = require('./simple_module.js');  
console.log(simpleModule);
```

Now, run the `import.js` file using the Node executable by running the following in a command-line prompt:

```
$ Node import.js
```

You should get the following output:

**ABC**

Let's see what we did here.

Inside the `simple_module.js` file, you declared that what you wanted to export from that module was the `ABC` string.

Then, in the `import.js` module, you imported that module by requiring it to use a relative path (`./simple_module.js`). If you give a path instead of a module name, Node.js will look for that file and try to load it. If you provide a relative path, Node.js will use that in relation to the current file, which in our case resolves to a file named `simple_module.js` inside the same directory as `import.js`.

Now let's create a folder named `lib` and create a new module in `lib/other_module.js` containing the following JavaScript code:

```
module.exports = 'DEF';
```

Also, replace the code in `import.js` with the following code:

```
var simpleModule = require('./simple_module.js');

console.log(simpleModule);

var otherModule = require('./lib/other_module.js');

console.log(otherModule);
```

You can see that we're importing the new module using the path relative to the current file.

You can run the `import.js` file again, like this:

```
$ Node import.js
```

It should output the following:

**ABCDEF**

You can also use two dots (`..`) to specify parent directories as you would in a normal relative path:

```
require('../module_a.js');
require('../folder/module_b.js');
```





You can also use absolute paths, such as `/path/to/my/module.js`, but since they're not generally portable between different computers, you should avoid them.

Also, you can omit the `.js` extension if you wish to. For instance, the `import.js` file code could be transformed to the following without resulting in any difference:

```
var simpleModule = require('./simple_module');

console.dir(simpleModule);

var otherModule = require('./lib/other_module');

console.dir(otherModule);
```

## Node Package Manager (NPM)

Node.js has a low-level API that is somewhat a translation and simplification of the Unix filesystem and networking API into JavaScript. If you plan to build a complex application, doing that solely on top of the core Node.js functionality can be hard and unproductive.

Fortunately, Node.js has a way of browsing, querying, installing, and publishing third-party modules into a central repository, and it's called NPM. NPM stands for Node Package Manager, and it consists of two things:

- ◆ A module repository that is fully browsable, accessible at <https://npmjs.org/>
- ◆ A command-line utility

The NPM repository contains a vast and growing collection of modules that were built by the community. Since that repository is fully browsable and searchable, you can use it to track down and inspect modules that may be interesting for building your application with.



Given the sheer number of available modules, a better source of module usefulness and quality can be the community forums and IRC channel, or even to inspect the code yourself.

In NPM, each packaged module has a name, which serves to uniquely identify that module. Each module can have one or more versions, and you can specify which one to install.

To start watching NPM in action, let's first create a `package.json` file. This file serves as a local package manifest, contains JSON (an object serialization format that is a subset of JavaScript), is placed at the root of your application folder and is, among other things, where you can declare what NPM modules your application depends on. Let's create one, then:

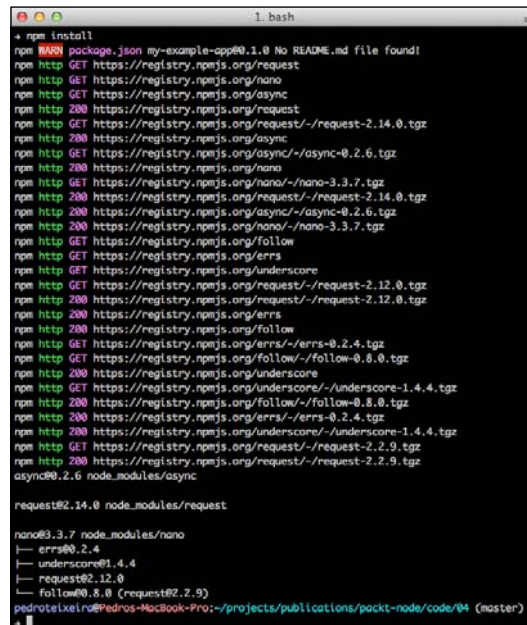
```
{
  "name": "my-example-app",
  "version": "0.1.0",
  "dependencies": {
    "request": "*",
    "nano": "3.3.x",
    "async": "~0.2"
  }
}
```

Once you have saved the contents of `package.json` in the current directory, run the following at the command line:

```
$ npm install
```

Once you have hit the *Enter* key, NPM will analyze the dependencies of your `package.json` file and try to download it from the Internet and install it.

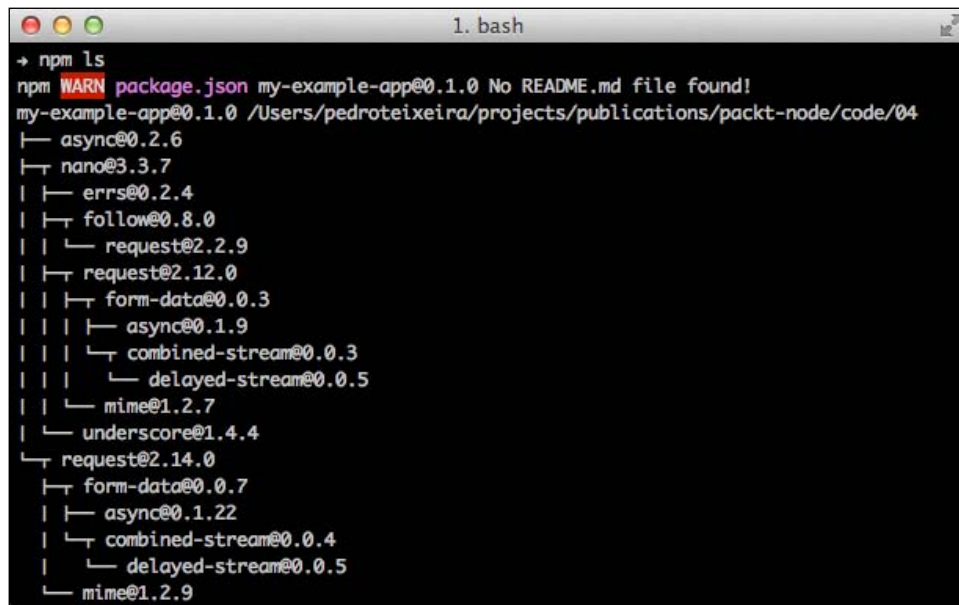
Once NPM finishes, you should have the `request`, `nano`, and `async` modules from NPM installed locally. You can then use the `npm ls` command to list the modules installed in the current directory.



```
1: bash
+ npm install
npm WARN package.json my-example-app@0.1.0 No README.md file found!
npm http GET https://registry.npmjs.org/request
npm http GET https://registry.npmjs.org/nano
npm http GET https://registry.npmjs.org/async
npm http 200 https://registry.npmjs.org/request
npm http GET https://registry.npmjs.org/request/-/request-2.14.0.tgz
npm http 200 https://registry.npmjs.org/async
npm http GET https://registry.npmjs.org/async/-/async-0.2.6.tgz
npm http 200 https://registry.npmjs.org/nano
npm http GET https://registry.npmjs.org/nano/-/nano-3.3.7.tgz
npm http 200 https://registry.npmjs.org/request/-/request-2.14.0.tgz
npm http 200 https://registry.npmjs.org/async/-/async-0.2.6.tgz
npm http 200 https://registry.npmjs.org/nano/-/nano-3.3.7.tgz
npm http GET https://registry.npmjs.org/follow
npm http GET https://registry.npmjs.org/errs
npm http GET https://registry.npmjs.org/underscore
npm http GET https://registry.npmjs.org/request/-/request-2.12.0.tgz
npm http 200 https://registry.npmjs.org/request/-/request-2.12.0.tgz
npm http 200 https://registry.npmjs.org/errs
npm http 200 https://registry.npmjs.org/follow
npm http GET https://registry.npmjs.org/errs/-/errs-0.2.4.tgz
npm http GET https://registry.npmjs.org/follow/-/follow-0.8.0.tgz
npm http 200 https://registry.npmjs.org/underscore/-/underscore-1.4.4.tgz
npm http 200 https://registry.npmjs.org/follow/-/follow-0.8.0.tgz
npm http 200 https://registry.npmjs.org/errs/-/errs-0.2.4.tgz
npm http 200 https://registry.npmjs.org/underscore/-/underscore-1.4.4.tgz
npm http GET https://registry.npmjs.org/request/-/request-2.2.9.tgz
npm http 200 https://registry.npmjs.org/request/-/request-2.2.9.tgz
async@0.2.6 node_modules/async

request@2.14.0 node_modules/request
├── errs@0.2.4
├── underscore@1.4.4
├── request@2.12.0
└── follow@0.8.0 (request@2.2.9)
pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)
+ 
```

This last command not only lists the NPM modules installed locally but also the modules that those modules depend on, recursively rendering it into this tree.



```
1. bash
→ npm ls
npm WARN package.json my-example-app@0.1.0 No README.md file found!
my-example-app@0.1.0 /Users/pedroteixeira/projects/publications/packt-node/code/04
├─ async@0.2.6
├─ nano@3.3.7
│   ├── errrs@0.2.4
│   ├── follow@0.8.0
│   ├── request@2.2.9
│   ├── request@2.12.0
│   ├── form-data@0.0.3
│   │   ├── async@0.1.9
│   │   ├── combined-stream@0.0.3
│   │   └── delayed-stream@0.0.5
│   ├── mime@1.2.7
│   └── underscore@1.4.4
└─ request@2.14.0
    ├── form-data@0.0.7
    ├── async@0.1.22
    ├── combined-stream@0.0.4
    ├── delayed-stream@0.0.5
    └── mime@1.2.9
```

The information written out by the NPM command line contains the version of each module. In the `package.json` manifest, you have a specification of which version you want to install, for each NPM module. For instance, we have used different version specifications such as `*`, `3.3.x`, and `~0.2`. These follow a convention named semantic versioning, which uses three integers to specify the version; it consists of the major version, minor version, and patch version numbers, all separated by a dot.

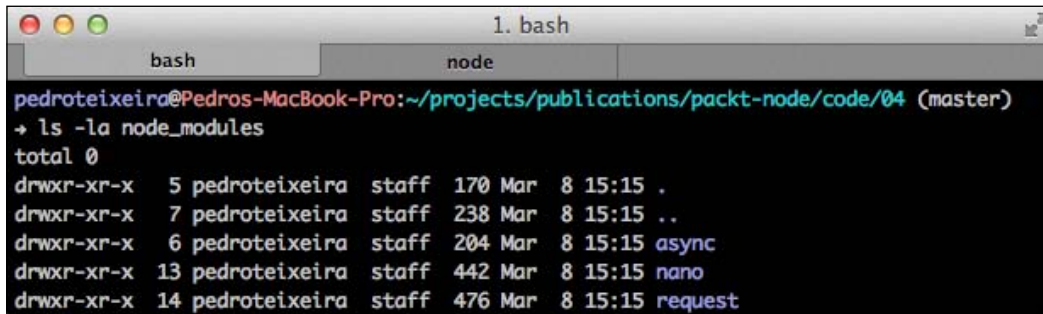
In the dependency list, you can specify the asterisk (`*`) as a wildcard that stands for any version. This says that your application depends on any version of that module. When you specify that you depend on Version `3.3.x` of the `nano` module, any patch version of Version `3.3` will satisfy your requirements. The `async` version specification (`~0.2`) has a similar effect, saying that your requirements are satisfied with any patch version of `0.2`. NPM will analyze what versions you specify and will install the latest available version that matches your specification, if available.

After having the dependencies installed, you can use them in your code. Here is an example of importing some modules that we may depend on:

```
var request = require('request');
var nano = require('nano');
var async = require('async');
```

Here is how Node.js resolves these dependencies when you require them.

By default, NPM installs the dependencies locally inside a directory named `node_modules`. Node.js will look for this directory when you use `require(<module name>)` and will look for the module there.



```

1. bash
bash node
pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)
→ ls -la node_modules
total 0
drwxr-xr-x  5 pedroteixeira  staff  170 Mar  8 15:15 .
drwxr-xr-x  7 pedroteixeira  staff  238 Mar  8 15:15 ..
drwxr-xr-x  6 pedroteixeira  staff  204 Mar  8 15:15 async
drwxr-xr-x 13 pedroteixeira  staff  442 Mar  8 15:15 nano
drwxr-xr-x 14 pedroteixeira  staff  476 Mar  8 15:15 request

```

If the module can't be found there, Node.js will keep looking for it in the `node_modules` directory of the parent directory if it exists, then keep climbing up the file system tree until the module is found or it has reached the root directory. This means that if you have a library inside a nested directory in your application, it can depend on a module installed at the root of your application. For instance, if you create a module inside `lib/db.js`, it can depend on `nano` and contain code like this:

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

## The callback pattern

Node.js uses the event-driven model of doing I/O, which means that every time that the current process has to talk to the filesystem or the network, it does so in a non-blocking manner.

The pseudo-code for typical blocking code when doing a remote call to a database may look something like this:

```
var result = query('SELECT * FROM articles');
console.log('result:', result);
```

The time that it takes between starting the query and getting back the result may take anything from a few milliseconds to a long time, and while that happens, your current program is just waiting. In order to be able to do more I/O at the same time, you need to introduce additional execution contexts, which typically happens in the form of threads or other processes.

In event-driven I/O, instead of returning the result of a remote operation, you pass a callback that gets invoked once that operation is finished. On such a platform, the equivalent of doing a database query similar to the previous blocking code would be:

```
query('SELECT * FROM articles', function(result) {
  console.log('result:', result);
});
```

Here, instead of relying on the return value, we're simply initiating the remote call and passing in a function callback as the last argument. When that remote call is completed, the function is invoked, passing in the result as the sole argument.

Actually, both the blocking and non-blocking versions have been simplified, since they don't predict the case when an error happens. The correct blocking version would then be:

```
try {
  var result = query('SELECT * FROM articles');
  console.log('result:', result);
} catch(err) {
  console.error('error while performing query:', err.message);
}
```

And the equivalent event-driven version would be:

```
query('SELECT * FROM articles', function(err, result) {
  if (err) {
    console.error('error while performing query:',
      err.message);
  } else {
    console.log('result:', result);
  }
});
```

In this last example, we're witnessing the Node.js callback pattern, which has two characteristics:

- ◆ **Callback last:** The callback is the last argument of the function that initiates I/O.
- ◆ **Error first:** The callback expects an error as the first argument and the results in the following arguments. If there is no error, the first argument is undefined or null. If an error exists, this object should be a JavaScript error instance.

Here is an actual working example of reading a file from disk, which you can write to a file named `read_file.js`:

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

fs.readFile(__filename, 'utf8', function(err, fileContent) {
  if (err) {
    console.error(err);
  } else {
    console.log('got file content:', fileContent);
  }
});
```

Here, we're using the `fs` module, and particularly, its `readFile` function to read the contents of the JavaScript source file from disk. We're also using the `__filename` variable, which contains the full path for the current source file. We specify the file path and the file encoding (UTF-8) and pass in a function callback as the last argument.

This last argument then gets called if there is an error or when the contents of the file are read from disk.

You can run this example in the command-line prompt like this:

```
$ Node read_file.js
```

It should output the content of the `read_file.js` file.

## Chaining I/O

When performing chaining, some I/O depends on the result of other I/O; you can chain the two operations by nesting the callback functions. For instance, if you want to verify the size of a file and only read it if it's smaller than 1 KB, you could write this into a file named `read_file_conditional.js`:

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

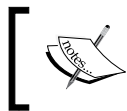
fs.stat(__filename, function(err, stats) {
  if (err) {
    console.error(err);
  } else {
    if (stats.size < 1024) {
      fs.readFile(__filename, 'utf8', function(err, fileContent) {
        {
          if (err) {
            console.error(err);
          } else {
            console.log('Got file content:', fileContent);
          }
        }
      });
    } else {
      console.log('Didn\'t read the file, it was too long.');
```

Alternatively, you can create the callback functions at the same level and name them as in this example (`read_file_conditional_unnested.js`):

```
var fs = require('fs');
function readFileCallback(err, fileContent) {
  if (err) {
    console.error(err);
  } else {
    console.log('Got file content:', fileContent);
  }
}

function statsCallback(err, stats) {
  if (err) {
    console.error(err);
  } else {
    if (stats.size < 1024) {
      fs.readFile(__filename, 'utf8', readFileCallback);
    } else {
      console.log('Didn\'t read the file, it was too long.');
```

This will have the benefit of reducing the code indentation level but makes the execution flow a bit harder to follow.



Alternatively, you can create or use an asynchronous flow control library such as `async`, which has some constructs that help you manage callback flow.

## Event emitter

The callback pattern is useful when you have I/O operations that have a clear end. If you have an object in which events happen throughout time, the callback pattern is not a good fit. Thankfully, Node.js has a built-in pattern named event emitter.

We have already seen objects that are event emitters such as the HTTP server object, which can emit a series of events throughout time. Each event has a type identified by a string. If you have an event emitter object, you can listen to any future occurrence of an event with a specific event type by binding a function like this:

```
function eventListener(a, b) {
  console.log('got event1', a, b);
}
emitter.on('event1', eventListener);
```

Here we're saying that every time we get an event with the `event1` type, the function named `eventListener` should be called. Each event type can emit the event with a given set of arguments that will be passed on to the listener. In this case, the listener is expecting two arguments, `a` and `b`.

Once you have a Listener function bound, you can remove it using `emitter.removeListener`, like this:

```
emitter.removeListener('event1', eventListener);
```

Besides these two listener management functions, you have also the `emitter.once` method available. Using this function, you can limit the number of callbacks to a maximum of one.

```
emitter.once('event1', eventListener);
```

You can also create an event emitter by using the Node.js event emitter API. Here is an example of an event emitter that emits a `tick` event every second:

```
var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

var count = 0;
setInterval(function() {
  emitter.emit('tick', count);
  count++;
}, 1000);

emitter.on('tick', function(count) {
  console.log('tick:', count);
});
```

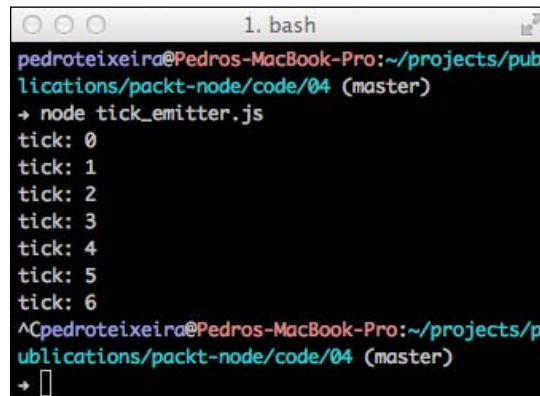
Here, we're using the `emitter.emit` method to, as the name says, emit an event. The first argument is the event type, and the remainder of the arguments will be passed on to the registered event listeners (if any).



If you save the code we just saw into a file named `tick_emitter.js`, you can run it like this:

```
$ Node tick_emitter.js
```

The console will show a new line every second before you cancel it (using `Ctrl + C`):

A terminal window titled '1. bash' showing the execution of a Node.js script. The prompt is 'pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)'. The user enters 'node tick\_emitter.js'. The output shows a series of 'tick: 0' through 'tick: 6' printed every second. The user then presses Ctrl+C, indicated by '^C', and the prompt returns to 'pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)' with a cursor on a new line.

```
pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)
→ node tick_emitter.js
tick: 0
tick: 1
tick: 2
tick: 3
tick: 4
tick: 5
tick: 6
^Cpedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)
→
```

## Parallelizing I/O

Parallelizing I/O is natural in Node.js; you don't need to spawn new threads of execution, simply start two or more I/O operations in parallel. In the following example, we want to make some HTTP requests in parallel to `http://www.twitter.com/`. For that, create a file named `http_requests_parallel.js` with the following content:

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

var urls = [
  'http://search.twitter.com/search.json?q=Node',
  'http://search.twitter.com/search.json?q=javascript'
];

var allResults = [];
var responded = 0;

function collectResponse(res) {
  var responseBody = '';
  res.setEncoding('utf8');

  /// collect the response body
  res.on('data', function(d) {
    responseBody += d;
  });
}
```

```
    /// when the response ends, we should have all the response
    body
    res.on('end', function() {
        var response = JSON.parse(responseBody);
        allResults = allResults.concat(response.results);
        console.log('I have %d results for',
            response.results.length, res.req.path);
        responded += 1;

        /// check if we have responses to all requests
        if (responded == urls.length) {
            console.log('All responses ended. Number of total
                results:', allResults.length);
        }
    });
}

urls.forEach(function(url) {
    http.get(url, collectResponse);
});
```

Here, we are using the `http` module to make `http.get` requests to the Twitter search API. We start out by declaring which URLs you want to make requests to and then declare a callback function named `collectResponse`. This function will be called when each of these requests is available. At this time, Node.js only has the response HTTP headers but doesn't yet have the response body. Because of that, we need to collect the body by listening to all the data events on the response that emits buffers; if you have the encoding set, it emits strings. Once the response ends, we can assume we have all the response body chunks and can finally parse it. The response is JSON-encoded, which means we can use the native `JSON.parse` function to transform the string into a JavaScript object, from which we then extract the `results` attribute.

Since this `collectResponse` function will be called for each individual request, we keep track of how many responses have ended. Once we know that we have all the responses, we say "all responses ended".

At the end, we kick off this whole process by starting the requests using the `http.get` function on each planned URL.

## Streams

Node has this built-in concept of streams, which comes from the philosophy of Unix systems. A stream represents a source of data read or written throughout time. For instance, you can read the entire contents of a file at once, but if the file is large, this process will take some time and will consume memory. Alternatively, you can read it as a stream, where you get a piece of the file content at a time.

## The readable streams

Here is an example where you open a file read stream and listen for data chunks from it:

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

var stream = fs.createReadStream('/path/to/large/file');

stream.on('readable', function() {
  var chunk;
  while(chunk = stream.read()) {
    console.log('got NPM data chunk of %d bytes',
      chunk.length);
  }
});

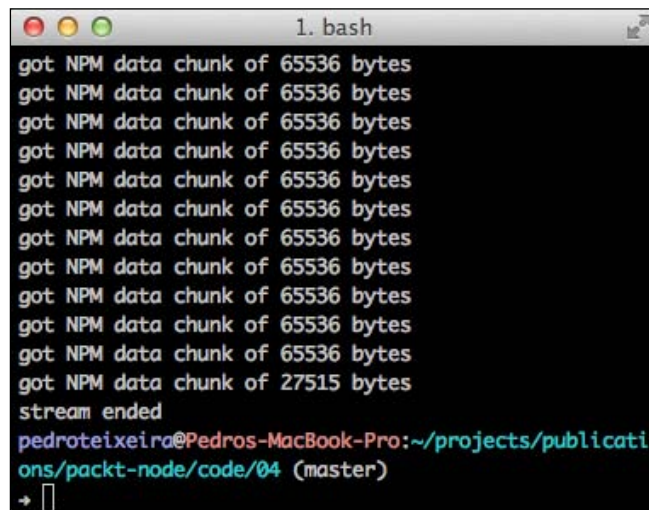
stream.once('end', function() {
  console.log('stream ended');
});
```

Here, you can see that a read stream is an event emitter that emits `readable` and `end` events, which you can listen to. When we get a `readable` event, we read from the stream until there is nothing more to read, and for each chunk of data, we print out its length. We're also logging `stream ended` to the console once we reach the end of the file.

If you save this code into a file named `file_stream.js`, you can run it like this:

```
$ Node file_stream.js
```

The console will then print out something like this:



```
1. bash
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 65536 bytes
got NPM data chunk of 27515 bytes
stream ended
pedroteixeira@Pedros-MacBook-Pro:~/projects/publications/packt-node/code/04 (master)
➔
```

Here, we can see that we get several data chunks exactly 64 KB in size, with the exception of the last one. We can also see that, after the stream ended, we didn't get any more data events.

## The writable streams

Besides being the source of data, a stream can alternatively or simultaneously be the target of data—a writable stream. You can write data out to a writable stream, and you can also end one. There are several examples of writable streams in Node.js. You can, for instance, create a writable file stream:

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

var stream = fs.createWriteStream(__dirname + '/out.txt');

var interval = setInterval(function() {
  stream.write('tick ' + Date.now() + '\n');
}, 100);

setTimeout(function() {
  clearInterval(interval);
  stream.end();
}, 4950);
```

Here, we're creating a file `WriteStream` and setting up two timers. The first timer is repetitive and fires every 100 milliseconds, and every time it fires, we compose and write out a string containing the current timestamp.

The second timer only fires one after 5 seconds, and once that happens, we cancel the repeating interval and end the write stream.

If you save this code example into a file named `writable_file_stream.js`, you can run it like this:

```
$ Node writable_file_stream.js
```

After 5 seconds, your Node process should end, and you should have a file named `out.txt` containing 931 bytes of text.

## The duplex streams

A stream can be both writable and readable at the same time, permitting both obtaining data from it and writing data to it. An example of such a stream can be a TCP connection:

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

var server = net.createServer();

server.on('connection', function(stream) {
```

```
stream.setEncoding('utf8');

console.log('got new stream');

stream.on('readable', function() {
  var buf;
  while(buf = stream.read()) {
    console.log('data: ', buf);
    stream.write(buf.toUpperCase());
  }
});

stream.on('end', function() {
  console.log('stream ended');
});

});

server.listen(8080);
```

Here, we're creating a TCP server, which we will bind to port 8080. On every connection we need to do the following:

- ◆ Set the string encoding so that we get UTF-8-encoded strings instead of buffer chunks when we read data from it.
- ◆ Add a listener for the `readable` events where we fetch all the available data from the stream. Every time we get a new data chunk, we do the following:
  - Print out the received data
  - Transform the string into uppercase
  - Write that string back into the stream

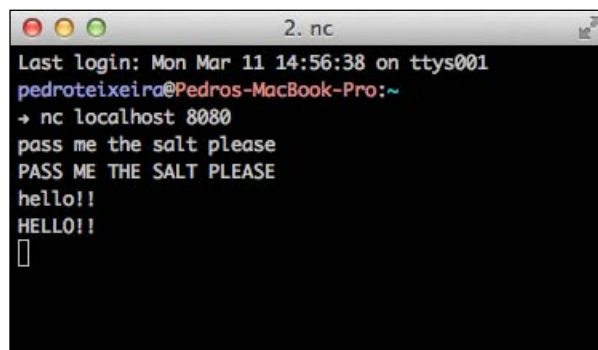
You can start this server by saving the previous code into a file named `tcp_server.js` and then starting it like this:

```
$ Node tcp_server.js
```

In a different console window, connect to the server using a command-line utility, such as Telnet or `nc`. Here, I'm using `nc` to connect to localhost port 8080:

```
$ nc localhost 8080
```

Now you can type text into the remote session, and the server should respond with that text in uppercase:



```
2. nc
Last login: Mon Mar 11 14:56:38 on ttys001
pedroteixeira@Pedros-MacBook-Pro:~
→ nc localhost 8080
pass me the salt please
PASS ME THE SALT PLEASE
hello!!
HELLO!!
█
```

## Stream flow control

In Node.js, streams are not only about sending and receiving data but also about controlling the flow of that data. When you write data into a writable stream, that operation is non-blocking, which means that the data may get stored somewhere before it's actually sent out to the underlying resource. For instance, if you're writing to a file or a network connection, Node.js may not be able to send that data into the kernel while the kernel buffers still hold previously sent data.

In Node.js streams, the call to `stream.write` will return a boolean value saying whether the buffer was flushed to the underlying resource. If the `write` operation returns `false`, the stream will later emit a `drain` event once the buffer gets flushed. We can test that by putting the following code in a file named `write_drain.js`:

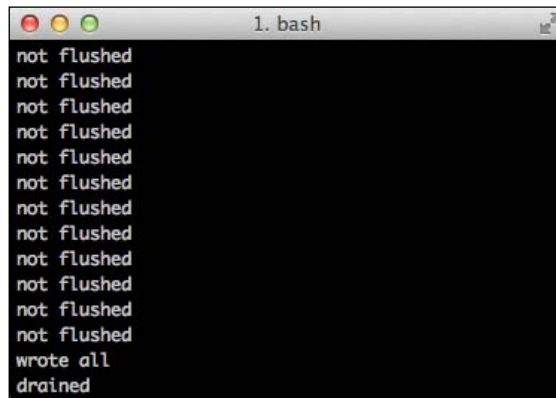
```
var fs = require('fs');
var stream = fs.createWriteStream(__dirname + '/out.txt');

for(var i = 0 ; i < 50000; i++) {
  if (! stream.write(i.toString() + ' ')) {
    console.log('not flushed');
  } else {
    console.log('flushed');
  }
}
console.log('wrote all');

stream.on('drain', function() {
  console.log('drained');
});
```

Then, run it:

```
$ Node write_drain.js
```

A terminal window titled "1. bash" with a black background and white text. It displays the output of the `write_drain.js` script. The output consists of 13 lines of "not flushed" followed by "wrote all" and "drained".

```
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
not flushed
wrote all
drained
```

You will observe several **not flushed** messages, followed by a **wrote all** message, followed by a **drained** message. This means that none of the `write` operations were immediately flushed; instead, they got queued. Once the Node manages to flush all that data, it emits a `drain` event.

Node.js also has API support for flow control on the `readable` streams. We have already observed that, when data is available on a `readable` stream, the `readable` event is emitted. This makes the data buffer up on the `readable` stream before we read it. As in the `writable` streams, as data gets buffered, memory gets consumed and will eventually run out. Fortunately, the `readable` streams have a built-in flow control mechanism as it ceases from pulling data from the underlying resource if a high watermark is reached. This high watermark, which can be configured in the `stream.Readable` constructor, defines the maximum number of bytes to store before pausing. Also, some concrete implementations of the `readable` streams allow you to change this value, as in the example of `fs.createReadStream`:

```
var fs = require('fs');
var stream = fs.createReadStream('/path/to/file/', {highWaterMark:
40000});
```

## Piping

If you have a `readable` and `writable` stream you can connect one to the other using the pipe mechanism:

```
var sourceStream = ...
var targetStream = ...
sourceStream.pipe(targetStream);
```

Here, every time there is data available on the source stream, it gets read and written to the target stream. Also, it is possible to unpipe a destination stream from the source stream using `stream.unpipe`:

```
sourceStream.unpipe(targetStream);
```

Next is an example of an echo server that pipes all the data coming from a connection back into that connection:

```
var net = require('net');
var server = net.createServer();

server.on('connection', function(stream) {
  stream.pipe(stream);
  stream.pipe(process.stdout);
});

server.listen(8080);
```

Here, we're setting up a TCP server listening on port 8080. Every time the server gets a new connection, we pipe the connection stream into itself and also into the process standard output. The first pipe has the effect of writing back every bit of data that any client sends, making this server effectively an echo server. The second pipe is writing every bit of data that one client sends into the process output, which is a writable stream.

If you save the previous code block in a file named `echo_server.js`, run it and connect it to a command-line utility such as `Telnet` or `nc`, you will notice that the client receives every bit of data that is sent to the server.

## The transform streams

Since the call to `stream.pipe` returns the target stream, it is possible to chain pipes:

```
var sourceStream = ...
var transformStream = ...
var targetStream = ...
sourceStream.pipe(transformStream).pipe(targetStream);
```

This last line pipes every bit of data coming in from `sourceStream` into `transformStream` and every bit of data coming out of `transformStream` into `targetStream`.

This can be useful when you need to react to incoming data and transform a stream into another stream. For instance, we can implement our previous uppercasing TCP server example into a Transform stream like this:

```
var Transform = require('stream').Transform;

var server = require('net').createServer();
server.on('connection', function(stream) {
```



```
stream.setEncoding('utf8');

var uppercase = new Transform({decodeStrings: false});
uppercase._transform = function(d, encoding, cb) {
  cb(null, d.toUpperCase());
};

stream.pipe(uppercase).pipe(stream);
});

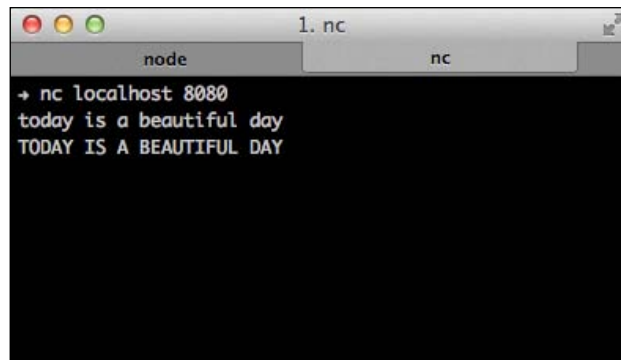
server.listen(8080);
```

Here, we're using the `stream.Transform` utility class from the Node core API to create a `Transform` stream. This stream has to implement a `_transform` method that performs the actual transformation. This function receives the callback as the third argument, which it has to call using the callback pattern: error first and result last. In our case, we can perform a callback immediately since the transformation is a synchronous operation, passing in `null` as the error argument to signal that no error occurred during the transformation.

You can save this last piece of code into a file named `uppercase_server_stream.js` and run it:

```
$ Node uppercase_server_stream.js
```

You can then, from another window, use `Telnet`, `nc`, or a similar command-line tool to open a connection to the server. In this example, I'm using `nc`:



These are the main concepts that will get you started in the Node.js world. The callback pattern, the event emitter, and the streams are the building blocks of the Node.js core API and also of an ecosystem of open-source modules contained in NPM. Since the Node.js core API is a low-level one, you will almost surely require using third-party modules to build an interesting application. With the knowledge of how these patterns work and with the help of some of the Node.js community resources, you will be able to understand and use them to accomplish your goals.

## People and places you should get to know

### Official sites

The official Node.js website, containing links to the downloads, API documentation, blog, and other interesting resources is as follows:

<http://Nodejs.org>

### Blogs

This is where the Node.js core development team posts updates and plans.

<http://blog.Nodejs.org>

### API documentation

The central resource for accessing documentation on the Node.js core API is as follows:

<http://Nodejs.org/api/>

### NPM modules

You can use it to search for packages and to show package documentation and metadata. The web frontend of the NPM repository is as follows:

<https://npmjs.org>

### Github repository

The versioned code repository where the Node.js core development happens is as follows:

<https://github.com/joyent/Node>

### Community

- ◆ Mailing list: <http://groups.google.com/group/Nodejs>

This is the mailing list for the Node.js core, where you can post questions that can be answered by the community.

- ◆ IRC channels are as follows:
  - **Server:** `irc.freeNode.net`
  - **Channel:** `#Nodejs`

This is a highly frequented IRC channel where you can post questions and answers about all things Node.js.

- ◆ Stack Overflow Node.js tag: <http://stackoverflow.com/questions/tagged/Node.js>  
All questions posted on Stack Overflow are tagged with Node.js.
- ◆ Nodeup podcast: <http://Nodeup.com>  
This has frequent podcasts hosted by the Node.js core team members and some Node.js community leaders.
- ◆ Nodetuts screencast tutorials: <http://Nodetuts.com>  
This has free screencast tutorials about Node.js.
- ◆ Node.js conferences:
  - Node Conf (USA): <http://www.Nodeconf.com>
  - Node Dublin (Ireland): <http://Nodedublin.com>

## Twitter

- ◆ Isaac Schlueter (the Node.js development leader):  
<https://twitter.com/izs>
- ◆ James Halliday (author of the many Node.js modules, guides, and products; co-creator of browserling and testling):  
<https://twitter.com/substack>
- ◆ Mikeal Rogers (organizer of the NodeConf; Node.js core team member; author of the request NPM module):  
<https://twitter.com/mikeal>
- ◆ Charlie Robbins (author of many Node.js NPM modules; CEO of Nodejitsu, a PaaS provider of Node.js):  
<https://twitter.com/indexzero>
- ◆ Paolo Fragomeni (author of many popular Node.js NPM modules):  
<https://twitter.com/hijlrx>
- ◆ Bert Belder (Node.js core and LibUV developer; co-founder of Strongloop, a Node.js):  
<https://twitter.com/piscisaureus>
- ◆ Tim Caswell (freelancer; author of many Node-related open source projects):  
<https://twitter.com/creationix>

- ◆ Daniel Shaw (co-founder of The Node Firm, a consulting company specializing in Node.js; an engineer working at Voxer):  
`https://twitter.com/dshaw`
- ◆ Elijah Insua (hardware hacker; author of the famous JSDOM NPM module):  
`https://twitter.com/tmpvar`
- ◆ Nuno Job (geek, open-source enthusiast, and shapes the future of Node.js at Nodejitsu):  
`https://twitter.com/dscape`
- ◆ Pedro Teixeira:  
`https://twitter.com/pgte`





Thank you for buying  
**Instant Node.js Starter**

## About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

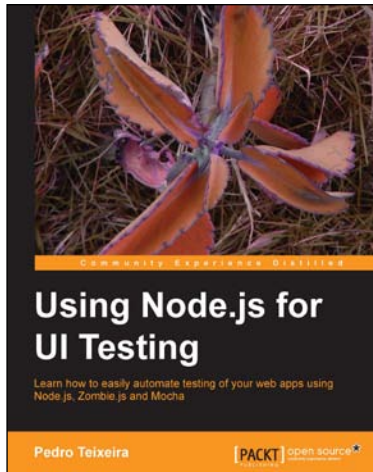
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

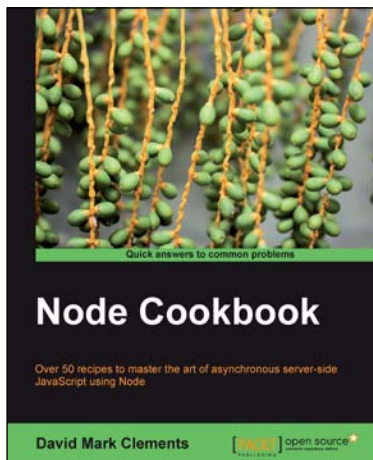


## Using Node.js for UI Testing

ISBN: 978-1-78216-052-6      Paperback: 146 pages

Learn how to easily automate testing of your web apps using Node.js, Zombie.js and Mocha

1. Use automated tests to keep your web app rock solid and bug-free while you code
2. Use a headless browser to quickly test your web application every time you make a small change to it
3. Use Mocha to describe and test the capabilities of your web app



## Node Cookbook

ISBN: 978-1-84951-718-8      Paperback: 342 pages

Over 50 recipes to master the art of asynchronous server-side JavaScript using Node

1. Packed with practical recipes taking you from the basics to extending Node with your own modules
2. Create your own web server to see Node's features in action
3. Work with JSON, XML, web sockets, and make the most of asynchronous programming

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

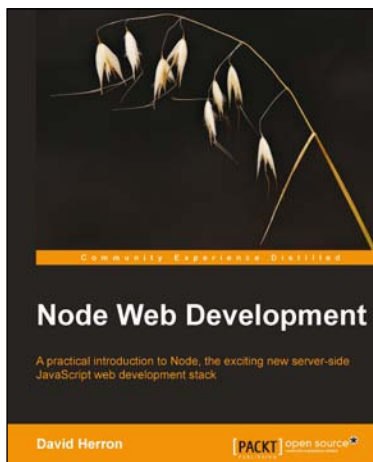


## Socket.IO Real-time Web Application Development

ISBN: 978-1-78216-078-6      Paperback: 140 pages

Build modern real-time web applications powered by Socket.IO

1. Understand the usage of various socket.io features like rooms, namespaces, and sessions
2. Secure the socket.io communication
3. Deploy and scale your socket.io and Node.js applications in production
4. A practical guide that quickly gets you up and running with socket.io



## Node Web Development

ISBN: 978-1-84951-514-6      Paperback: 172 pages

A practical introduction to Node, the exciting new server-side JavaScript web development stack

1. Go from nothing to a database-backed web application in no time at all
2. Get started quickly with Node and discover that JavaScript is not just for browsers anymore
3. An introduction to server-side JavaScript with Node, the Connect and Express frameworks, and using SQL or MongoDB database back-end

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles