

Node.js

Succinctly[®]

by Emanuele DelBono

Node.js Succinctly

By

Emanuele DelBono

Foreword by Daniel Jebaraj



Copyright © 2016 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Iimportant licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Stephen Haunts

Copy Editor: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	7
About the Author	9
About this Book	10
Chapter 1 An Introduction to Node.js	11
The history	11
How to obtain Node.....	13
Chapter 2 Hello Node.js	16
Hello world	16
Event driven.....	17
Single thread	17
Non-blocking I/O	18
The event loop.....	18
The Node.js runtime environment.....	21
Event emitters	23
Callback hell	25
Summary	27
Chapter 3 The Node.js Ecosystem.....	28
The module system	28
Summary	32
Chapter 4 Using the Filesystem and Streams.....	33
The fs module	33
Reading a file.....	33
Writing a file	33
Watching files	34

The path module	34
Streams.....	35
Readable streams.....	36
Writable streams.....	37
Summary	37
Chapter 5 Writing Web Applications.....	38
The http module	38
Express.js.....	39
Jade.....	42
Middleware	44
Chapter 6 Real-Time Apps with WebSocket.....	47
WebSocket.....	47
Socket.IO	47
Chapter 7 Accessing the Database.....	52
Accessing PostgreSQL.....	52
Accessing MongoDB	56
Chapter 8 Messaging with RabbitMQ	60
Why is RabbitMQ useful?	62
Chapter 9 Support Tools: Build and Testing.....	63
Mocha and Chai	63
Gulp	66
ESLint	68
Appendix A: Introduction to ES6	71
Arrow functions.....	71
const and let.....	72
Template strings.....	73

Classes	73
Destructuring assignment.....	75
Default parameters.....	75
Rest and spread.....	76
Summary.....	76

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Emanuele DelBono ([@emadb](#)) is a web developer based in Italy. He is one of the founders of CodicePlastico, a software house that builds web applications for its customers. He architects and develops web applications in Node.js and .NET.

Emanuele is also a speaker at various conferences about web development and agile practices. He plays an active role in Italian development communities such as Webdebs.org.

About this Book

This book is a quick view on Node.js, this new, astonishing, popular, fresh technology that is spreading through developers all around the world. This is a good introduction to the platform that inspects various topics from basic to intermediate. It also provides an overview on the environment around the platform so that after this book, the readers should have a clear idea about what they can do with Node and how to start using it. To read and fully understand this book, you need to know JavaScript. Syncfusion has an e-book titled *JavaScript Succinctly* that can be used to build the necessary skills to follow the examples in this book. You can download the e-book from the [Syncfusion website](#).

Chapter 1 An Introduction to Node.js

The history

Node.js is the platform that, in a very short time, has reached such worldwide success that today it is one of the most interesting platforms for writing web applications and more.

Node.js was born in 2009 from an idea of Ryan Dahl, who was searching for a way to track the time needed to upload a file from a browser without continuously asking the server “how much of the file is uploaded?” His idea was to explore what would happen if the requests were non-blocking, and JavaScript seemed the perfect language for two main reasons: at the time, it didn’t have an I/O library, and the async pattern, which is very useful for writing non-blocking applications, was already present in the language.



Figure 1: The Node.js website of 2009

So he took the Google Chrome V8 engine and used it as a base for writing a basic event loop and a low-level I/O API. He presented that embryo of what was to become Node.js at the European JsConf on November 8, 2008, and suddenly the project received a lot of interest from the community.

In 2011, the first version of the npm package manager was published and developers started to publish their own libraries to the npm registry. npm is surely one of the keys for the success of Node.js. npm makes it very easy for everyone to use and publish libraries and tools.



Figure 2: The first logo

The first releases of Node were not so easy to use. Breaking changes happened quite often from one version to another, at least until 2012 with the 0.8 version.

An interesting thing about the community is that since the very first 0.something edition, they started to develop frameworks and tools for writing complex applications. The first version of Express is from 2010, Socket.IO 2010, Hapi.js 2012. This means the community gave a lot of trust from the beginning, to the point that in 2012, LinkedIn and Uber started using it in production.

From 2011 to 2014, Node.js was gaining a lot of consensus, and even though Ryan Dahl left the project in 2012, more and more developers joined the platform, leaving Ruby, Java, Python, and C# in favor of Node.

At the end of 2014, some of the committers were not happy about the governance of the project that, until then, had been headed by Joyent, the company for which Ryan Dahl had worked at the time of creation. They decided to fork the main repository and created IO.js with the intent of providing faster release cycles and staying more up-to-date with the API and performance improvements of V8.

Those months were something strange, because a lot of developers didn't know what would happen in the future with two codebases that could diverge, making it difficult to choose which one to use.

A few months later, Joyent, with other companies (IBM, Microsoft, PayPal, Fidelity, SAP, and The Linux Foundation), decided to support the Node.js community with an open governance like the developers of IO.js wanted. This move made it possible for the two forks of Node.js to merge under the governance of the Node.js Foundation.

The Node.js Foundation's mission is to enable the adoption and help the development of Node.js and other related modules through an open governance model that encourages participation, technical contribution, and a framework for long-term stewardship by an ecosystem invested in Node.js's success.

With the advent of the Node.js Foundation, the first real, stable version of Node was released: the new version “1.0” of Node.js, 4.0.0.

A few months later, in late 2015, the Long Term Support (LTS) plan was released with version 4.2.0. This 4.x version will be supported for 30 months. In parallel, other versions will be released without the LTS (at the time of writing, version 5.10.0 is out). LTS implies that big companies are more lean to adopt Node.js as a development platform.

So today, Node.js is here to stay and has nothing less than other web platforms. With the Foundation, its future is surely bright and new improvements will continue to come.

Big companies like Microsoft, which was actively involved in porting Node.js on Windows, IBM, PayPal, Red Hat, Intel, and its inventor Joyent are betting on and investing a lot in Node.

The community was and still is one of the winning points of Node.js. The Node community is very inclusive and active, and in a few years they have produced more than 200,000 libraries and tools to use with Node.js.

Obtaining help and participating in the community is quite easy. The starting point is the [GitHub website](#) where you can find the source code and read about its issues. Committers are used to discussing the issues with the community. There is also an IRC channel #Node.js on [irc.freenode.net](#). If you prefer Slack, there is a channel on [Nodejs.slack.com](#). There is also the [Google group](#) and newsletters, tags on [StackOverflow](#), and more. You can be sure to find what you need.



Note: We cannot introduce Node.js without spending some words on Chrome V8. V8 is the engine that, so far, is behind the execution of Node.js applications. It was originally developed by Google as the JavaScript engine that runs the Chrome browser. Since all the code was released open source and since the ideas inside V8 were quite innovative, the Node.js developers find it natural to use V8 as the engine to run JavaScript code on the server.

How to obtain Node

Node.js is available on its website, [nodejs.org](#). I won't spend time explaining how to install it on the various platforms since on the official website, in the download section, there are installers for every platform, with instructions on how to install it. After the installation is completed, you have on your computer the Node executable (a note for Windows users: yes, you have to use the command line!) and the npm (Node Package Manager) that we will see later.



Node.js® is a JavaScript runtime built on **Chrome's V8 JavaScript engine**. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, **npm**, is the largest ecosystem of open source libraries in the world.

Download for OS X (x64)

v4.2.6 LTS
Mature and Dependable


v5.5.0 Stable
Latest Features

Other Downloads | Changelog | API Docs

Other Downloads | Changelog | API Docs

Figure 3: The current Nodejs.org home page

To test that everything is installed correctly, you can type **node** in the command line to enter the REPL interface. Once inside, you can try to execute some JavaScript code.

 **Note:** *REPL stands for Read Evaluate Print Loop. It is a quite common tool in Ruby and Node environments and is often used to learn a library or just to try some code. It consists of a command line executable (node in this case). You can run it by typing node in the terminal, which will give you access to the REPL. Here, you type any valid JavaScript statement and the REPL evaluates it for you immediately.*

```
1. node (node)
Last login: Mon May 23 20:16:38 on ttys000
~ node
> function sum(a, b) {
...   return a + b
... }
undefined
> sum(3,4)
7
>
```

Figure 4: Using the REPL

The Node REPL is a useful tool to test some code or to play with some library. Windows users are not used to tools like this, but I invite them to play with it to discover how useful it is.

To write a real-world application, you can use the text editor that you prefer, like [Vim](#), [SublimeText](#), or [Atom](#). If you want something more, [Visual Studio Code](#) is one the best lightweight IDEs for Node.js, and even if it is very similar to Atom (both are built on Electron), VS Code adds an integrated debugger that is very useful to inspect the application state during the run phase.

If VS Code is too lightweight, you can use [Visual Studio](#) with the Node.js tools installed or [WebStorm](#), a full IDE suited for web applications.

Chapter 2 Hello Node.js

Now that we have installed all the things that we need to start using Node.js, it's time to write a small program to investigate what Node is, what its peculiarities and advantages over other platforms are, and how to use it best to develop applications.

Hello world

The basic hello world application in Node.js is something like this:

Code Listing 1

```
console.log('Hello world');
```

It's not much. It's just a JavaScript instruction. But if we write that line inside a file named `index.js`, we can run it from the command line. We can create a new file called `index.js`, and inside it we write the previous statement. We save it and we type this in the same folder:

Code Listing 2

```
> node index.js
```

And what you obtain is the “Hello world” string in the terminal. Not so interesting. Let's try something more useful.

In a few lines of code, we can create the HTTP version of the hello world:

Code Listing 3

```
const http = require('http')
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/plain'})
  response.end("Hello World")
})
server.listen(8000)
```

This is a basic web server that responds “Hello World” to any incoming request. What it does is require from an external library the **http** module (we will talk about modules in the next chapter), create a server using the **createServer** function, and start the server on the port 8000.

The incoming requests are managed by the callback of the **createServer** function. The callback receives the **request** and **response** objects and writes the header (status code and content type) and the string **Hello World** on the **response** object to send it to the client.

We can run this mini HTTP server from the command line. Just save the preceding code inside a file named `server.js` and execute it from the command line like we did in the previous example:

Code Listing 4

```
> node server.js
```

The server starts. We can prove it by opening a browser and going to <http://localhost:8000> to obtain a white page with Hello World on it.



Figure 5: Hello World in the browser

Apart from the simplicity, the interesting part that emerges from the previous code is the asynchronicity. The first time the code is executed, the callback is just registered and not executed. The program runs from top to bottom and waits for incoming requests. The callback is executed every time a request arrives from the clients.

This is the nature of Node.js. Node.js is an event-driven, single-thread, non-blocking I/O platform for writing applications.

What does that mean?

Event driven

In the previous example, the callback function is executed when an event is received from the server (a client asks for a resource). This means that the application flow is determined by external actions and it waits for incoming requests. In general events, when something happens, it executes the code responsible for managing that event, and in the meantime it just waits, leaving the CPU free for other tasks.

Single thread

Node.js is single thread; all your applications run on a single thread and it never spawns on other threads. From a developer point of view, this is a great simplification. Developers don't need to deal with concurrency, cross-thread operations, variable locking, and so on. Developers are assured that a piece of code is executed at most by one single thread.

But the obvious question to be asked is: how can Node be a highly scalable platform if it runs on a single thread?

The answer is in the non-blocking I/O.

Non-blocking I/O

The idea that lets Node.js applications scale to big numbers is that every I/O request doesn't block the execution of an application. In other words, every time the application accesses an external resource, for example, to read a file, it doesn't wait for the file to be completely read. It registers a callback that will be executed when the file is read and in the meantime leaves the execution thread for other tasks.

This is the reason why a single thread is sufficient to scale: the application flow is never blocked by I/O operations. Every time an I/O happens, a callback is registered on a queue and executed when the I/O is completed.

The event loop

At the heart of the idea of non-blocking I/O is the event loop. Consider the previous example of a simple web server. What happens when a request arrives before the previous one was served? Remember that Node.js is single thread, so it cannot open a new thread and start to execute the code of the two requests in parallel. It has to wait, or better yet, it puts the event request in a queue and as soon as the previous request is completed it dequeues the next one (whatever it is).

Actually, the task of the Node engine is to get an event from the queue, execute it as soon as possible, and get another task. Every task that requires an external resource is asynchronous, which means that Node puts the callback function on the event queue.

Consider another example, a variation of the basic web server that serves a static file (an `index.html`):

Code Listing 5

```
var http = require('http')
var fs = require('fs')
var server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'});
  fs.readFile('./index.html', (err, file) => {
    response.end(file);
  })
})
server.listen(8000)
```

In this case, when a request arrives to the server, a file must be read from the filesystem. The **readFile** function (like all the async functions) receives a callback with two parameters that will be called when the file is actually read.

This means that the event “the file is ready to be served” remains in a queue while the execution continues. So, even if the file is big and needs time to be read, other requests can be served because the I/O is non-blocking (we will see that this method of reading files is not the best one).

When the file is ready, the callback will be extracted from the queue and the code (in this case the function **response.end(file)**) will be executed.

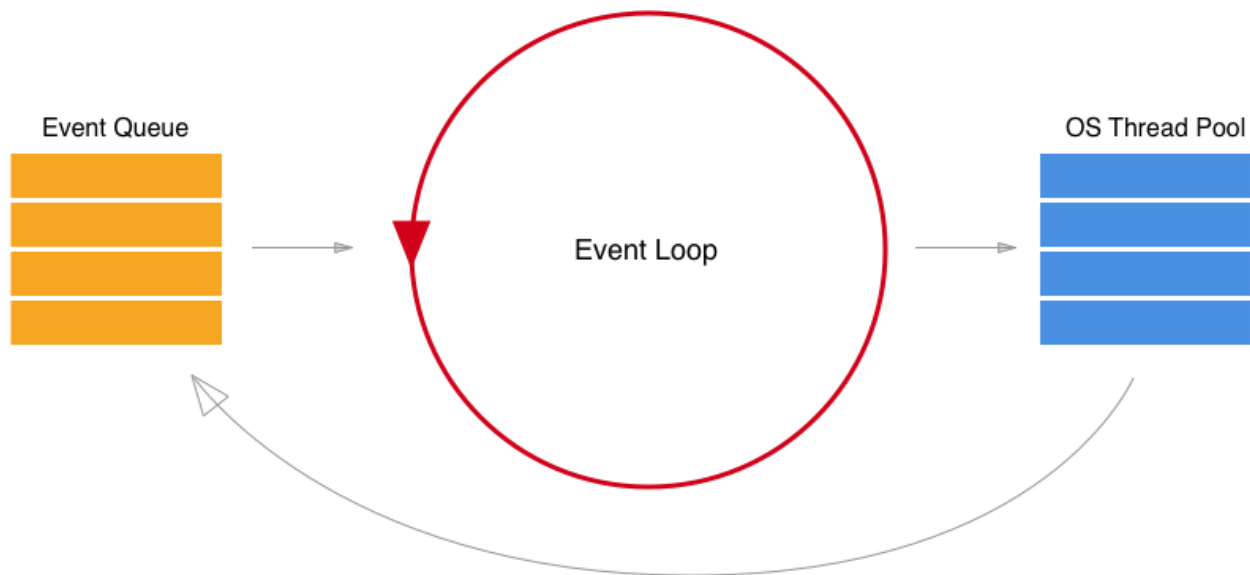


Figure 6: The Node.js event loop

The event loop is the thing that continues to evaluate the queue in search of new events to execute.

So the fact that Node is single-thread simplifies a lot of the development, and the non-blocking I/O resolves the performance issues.

The event loop has various implications on how we write our code. The first and most important is that our code needs to be as fast as possible to free the engine so that other events can be served quickly.

Consider another example. You should already be familiar with the JavaScript function **setInterval**. It executes a callback function every specified number of milliseconds.

Code Listing 6

```
setInterval(() => console.log('function 1'), 1000)
setInterval(() => console.log('function 2'), 1000)

console.log('starting')
```

When we run this code, the output will be something like this:

Code Listing 7

```
starting
function 1
function 2
```

```
function 1  
function 2
```

What happens inside?

The first line adds in the queue the callback that writes “function 1” to the console after one second. The second line does the same and the writes “function 2.”

Then it writes “starting” to the console.

After about one second, “function 1” will be printed to the console, and just after “function 2” will appear. We can be sure that function 1 will be printed before function 2 because it is declared first, and so it appears first in the queue.

So this program continues to print the two functions to the console.

Now we can try to modify the code:

Code Listing 8

```
setInterval(() => console.log('function 1'), 1000)  
setInterval(() => {  
  console.log('function 2')  
  while (true) { }  
}, 1000)  
console.log('starting')
```

We are simulating a piece of code that is particularly slow...infinitely slow!

What happens when we run this script? When it is its turn, function 2 will be executed and it will never release the thread, so the program will remain blocked on the while cycle forever.

This is due to the fact that Node is single-thread and if that thread is busy doing something (in this case cycling for nothing), it never returns to the queue to extract the next event.

This is why it is very important that our code is fast, because as soon as the current block finishes running, it can extract another task from the queue. This problem is solved quite well with asynchronous programming.

Suppose that at a certain point we need to read a big file:

Code Listing 9

```
var fs = require('fs')  
var data = fs.readFileSync('path/to/a/big/file')
```

If the file is very big, it needs time to be read, and since this piece of code is synchronous, the thread is blocked while waiting. All the data and all the other events in the queue must await the completion of this task.

Fortunately, in Node.js all the I/O is asynchronous and instead of using the **readFileSync**, we can use the async version **readFile** with a callback:

Code Listing 10

```
var fs = require('fs')
fs.readFile('path/to/a/big/file', (err, data) => {
  // do something with data
})
```

Using the async function guarantees that the execution continues to the end, and when the file is eventually read and the data ready, the callback will be called, passing the read data. This doesn't block the execution and the event loop can extract other events from the queue while the file is being read.



Note: *It's an accepted pattern that the first parameter of a callback function is the possible error.*

The Node.js runtime environment

When we run a script using **node index.js**, Node loads the `index.js` code and after compiling it, Node executes it from top to bottom, registering the callbacks as needed.

The script has access to various global objects that are useful for writing our applications. Some of them are:

Table 1: Global objects and functions

__dirname	The name of the folder that the currently executing script resides in.
__filename	The filename of the script.
console	Used to print to standard output.
module	A reference to the current module (more on this later).
require()	Function used to import a module.

As already stated, Node comes with a REPL that is accessible by running Node from the command line.

Inside the REPL we can execute JavaScript code and evaluate the result. It is also possible to load external modules by calling the `require` function. It is very useful for testing and playing with new modules to understand how they work and how they have to be used.

The Node.js REPL supports a set of command line arguments to customize the experience:

```
~/ $ node --help
```

```
Usage: node [options] [ -e script | script.js ] [arguments]
```

```
node debug script.js [arguments]
```

Options:

<code>-v, --version</code>	print Node.js version
<code>-e, --eval script</code>	evaluate script
<code>-p, --print</code>	evaluate script and print result
<code>-c, --check</code>	syntax check script without executing
<code>-i, --interactive</code>	always enter the REPL even if stdin does not appear to be a terminal
<code>-r, --require</code>	module to preload (option can be repeated)
<code>--no-deprecation</code>	silence deprecation warnings
<code>--throw-deprecation</code>	throw an exception anytime a deprecated function is used
<code>--trace-deprecation</code>	show stack traces on deprecations
<code>--trace-sync-io</code>	show stack trace when use of sync IO is detected after the first tick
<code>--track-heap-objects</code>	track heap object allocations for heap snapshots
<code>--v8-options</code>	print v8 command line options
<code>--tls-cipher-list=val</code>	use an alternative default TLS cipher list
<code>--icu-data-dir=dir</code>	set ICU data load path to dir (overrides <code>NODE_ICU_DATA</code>)

Environment variables:

NODE_PATH	'-separated list of directories prefixed to the module search path.
NODE_DISABLE_COLORS	set to 1 to disable colors in the REPL
NODE_ICU_DATA	data path for ICU (Intl object) data
NODE_REPL_HISTORY	path to the persistent REPL history file

Documentation can be found at <https://nodejs.org/>.

Event emitters

To write asynchronous applications, we need the support of tools that enable the callback pattern. In the Node standard library, there is the **EventEmitter** module that exposes the functionality to implement an observer.

Let's try a first example:

Code Listing 11

```
const EventEmitter = require('events').EventEmitter
let emitter = new EventEmitter()
emitter.on('newNumber', n => console.log(n * 2))
for (let i = 0; i < 10; i++) {
  emitter.emit('newNumber', i)
}
```

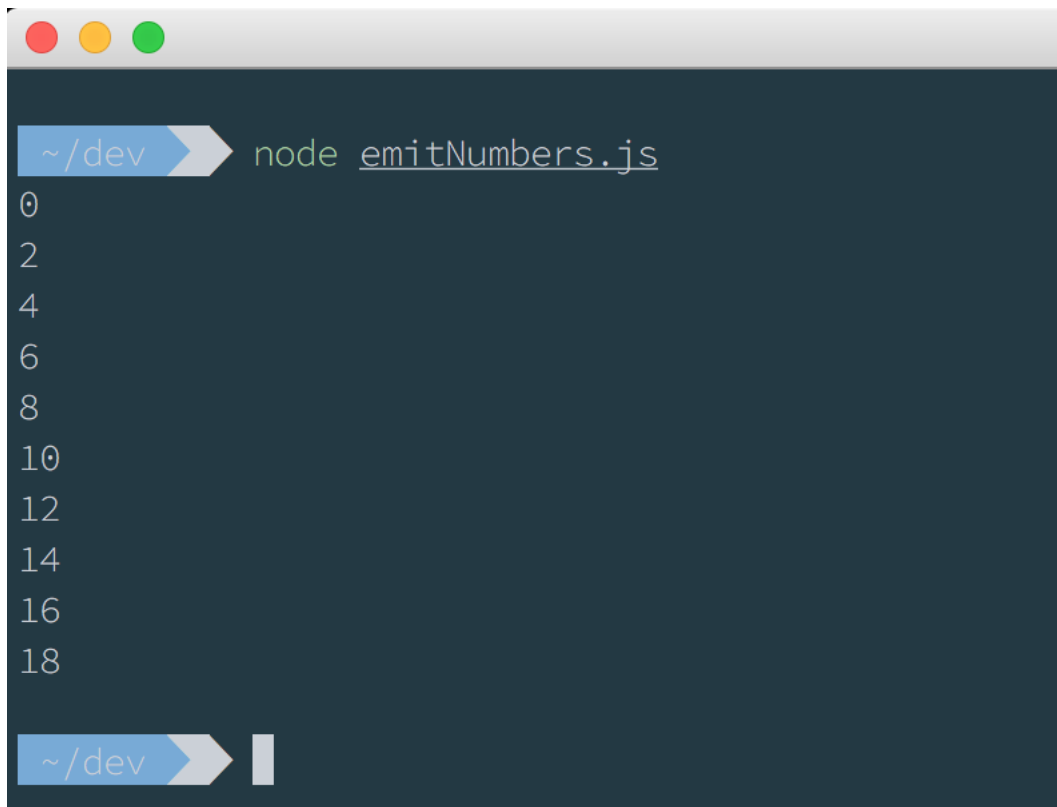
This small program implements a function that doubles the number from zero to ten using an observer pattern through the **EventEmitter**. Even if the program is not very useful, it explains the use of **EventEmitter**.

The first line loads the **events** module and gets the **EventEmitter** function that is used to create a new emitter.

The even emitter object has two main methods: **emit** and **on**.

on is used to subscribe to a particular event. It receives an arbitrary event name (**newNumber** in this case) and a callback function that will be executed when the event occurs.

If we execute the code, we obtain a list of numbers on the terminal. But what is the flow of the program? **EventEmitter** is synchronous, and that means the callback is called on every cycle of the **for** loop. If we add some console messages to log the execution, we will see that after emitting **newNumber**, the callback is executed immediately.



```
~/dev ➤ node emitNumbers.js
0
2
4
6
8
10
12
14
16
18
~/dev ➤
```

Figure 7: Event emitter

How can we transform this code to be asynchronous?

We can use the `setImmediate` function that, even if it is called immediate, puts the event in the queue and continues with the execution.



Note: There is some confusion around the `setImmediate` and `process.nextTick` functions. They seem similar (and they are) but they have a small, subtle difference. `setImmediate` puts the function in the queue, so if there are other functions in the queue, the `setImmediate` function will be executed after these other functions. `process.nextTick` puts the function at the head of the queue so that it will be executed exactly at the next tick, bypassing other functions in the queue.

The code is transformed:

Code Listing 12

```
const EventEmitter = require('events').EventEmitter
let emitter = new EventEmitter()
emitter.on('newNumber', n => setImmediate(() => console.log(n * 2)))
for (let i = 0; i < 10; i++) {
  emitter.emit('newNumber', i)
}
```


Every time the emitter emits an event, the listener calls the **setImmediate** function with the callback to execute. That callback is queued and executed after the current piece of code is complete (at the end of the **for** loop).

This is a naïve implementation of an async flow, but the important thing is to understand and become confident with these patterns because as Node developers, we have to know very well how the event loop works and how we can manage async calls.

Callback hell

The event loop is a great way to leave our program free to continue while the operating system completes the operation we need, but at the same time, serializing different tasks using this pattern becomes tedious and hard to maintain.

Suppose that we have three different async tasks that have to be done in a series. The first task concatenates two strings, the second uppercases a string, and the third decorates the string with stars. We assume that these three tasks are asynchronous:

Code Listing 13

```
function concat(a, b, callback){
  setTimeout(function(){
    var r = a + b
    callback(r)
  }, 0)
}

function upper(a, callback){
  setTimeout(function(){
    var r = a.toUpperCase()
    callback (r)
  }, 0)
}

function decor(a, callback){
  setTimeout(function(){
    var r = '*' + a + '*'
    callback (r)
  }, 0)
}
```

Since they are asynchronous, the signature of the function receives a callback that has to be called at the end. To simulate the asynchronicity, we have used the **setTimeout** function.

To serialize the call to these three functions, we can do this:

Code Listing 14

```
concat('hello', 'world', r1 => {  
  upper(r1, r2 => {  
    decor(r2, r3 => {  
      console.log(result, r3) // *HELLOWORLD*  
    })  
  })  
})
```

When the code assumes this shape of a Christmas tree, we are in a case of "callback hell." In this example we have three levels, but sometimes the levels are four, five, or more.

To better understand how the callbacks are called, take a look at the following image:

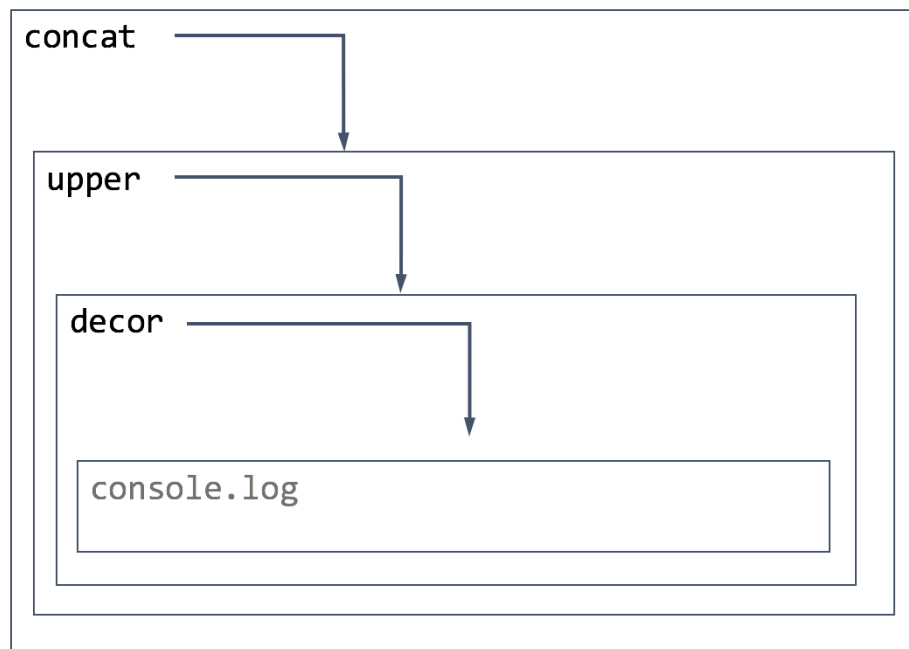


Figure 8: Callbacks

Every function is called with two parameters: an input variable and a callback, the next function to call. So each function does something and calls the next one.

To solve this issue, ECMAScript 6 introduced the concept of promises. A promise is a function that will succeed or fail. I used *will* because the success or failure is not determined at the time of execution, since our code is asynchronous.

Let's see how to rewrite the last piece of code using promises. First of all, we need to transform the three functions into promises:

Code Listing 15

```
function concatP(a, b) {
  return new Promise(resolve => {
    concat(a,b, resolve)
  })
}

function upperP(a) {
  return new Promise(resolve => {
    upper(a, resolve)
  })
}

function decorP(a) {
  return new Promise(resolve => {
    decor(a, resolve)
  })
}
```

Now we have three functions that return a promise. As already stated, the promise will resolve sometime in the future. It will be resolved when the function **resolve** is called.

After this refactoring, we can call the three functions in sequence using this pattern:

Code Listing 16

```
concatP('hello', 'world')
  .then(upperP.bind(this))
  .then(decorP.bind(this))
  .then(console.log) // *HELLOWORLD*
```

This code is not nested like in Code Listing 14, and it became more readable.

Summary

In this chapter, we started to touch the basics of the world of Node.js. We learned the tenets of a Node.js program, the simplicity of the single thread, and the peculiarity of the event loop and the asynchronous I/O. Now we know how to run a simple script or play with the REPL. In the next chapters we go deeper, and start to learn how to write bigger programs using modules and other useful tools.

Chapter 3 The Node.js Ecosystem

Node.js is quite young, especially compared to other platforms like PHP, .NET, or Ruby, but since its usage became so widespread in so few years and lots of developers moved to Node from other platforms, the community has created tons of libraries, tools, and frameworks. It's like the experienced developers from other environments bring to Node the best from these other worlds.

The module system

In [Chapter 2](#), we wrote some very basic programs and used a couple of external modules to access the filesystem and use the **EventEmitter**.

We used the **require** function to “import” the functionality defined in module **fs** in our program. The Node module system is based on the [CommonJS](#) specification. **The `require` function evaluates the code defined in the specified module and returns the `module.exports` object.** This object can be, and usually is, used to return something to the caller.

To understand how it works, let's start with a basic module.

Code Listing 17

```
// file: greeter.js
module.exports = (who) => {
  console.log(`Hello ${who}`)
}
```

We can use this module:

Code Listing 18

```
// file: index.js
var greet = require('./greeter')
greet('ema')
```

In **greeter.js** we assign to the object **module.exports** a function with one parameter. That function simply prints to the console.

module.exports is a global object that is exported to the caller using the **require** function. The **require** function returns an object that references **module.exports** of that file.

Everything defined inside a module is private and doesn't pollute the global scope except what is assigned to `module.exports`, and when a module is required, it is cached for better performance. This means the subsequent requires to the same module receive the same instance (the code inside the module is executed only once).

In the first example, we exported a single function. There are no limits to what we can export. We can export objects, classes, or functions.

Code Listing 19

```
// file: export_object.js
module.exports = {
  name: 'emanuele',
  surname: 'delbono',
  getFullName: function(){ return `${this.name} ${this.surname}` }
}
```

In this case, we are exporting a literal object with two properties (name and surname) and a function **getFullName**.

To use this module:

Code Listing 20

```
// file: index.js
var user = require('./export_object')
console.log(user.name); // emanuele
console.log(user.getFullName()); //emanuele delbono
```

To require a module, we simply need to invoke the **require** function. In a case where we are requiring a module that is written by us and is not in the **node_module** folder, we must use the relative path (in this case the module and the main program are in the same folder). If we are requiring a system module or a module downloaded using **npm**, we simply specify the module name (in Chapter 2 we required **fs** by just specifying its name).

As stated before, the required modules are cached for future use. This is primarily for performance reasons and is generally a good thing. Sometimes we need the content of the module executed every time it is requested.

One trick is to export a function and call that function. The function implementation is cached, but every time we execute it we can obtain something new.

For example:

Code Listing 21

```
// file: export_object.js
module.exports = function(){
  return { executionTime: new Date() }
}
```

This module exports a function that returns an object. Every time we execute the function, we obtain a new object, eventually with different properties.

How can we start using external modules?

We already talked about npm in [Chapter 1](#). npm is the package manager for Node.js applications and we use it to install packages that we can use to develop our applications.

npm is also the registry in which the components are stored and indexed. On <http://www.npmjs.com/> we can search for packages and find information about their usage.

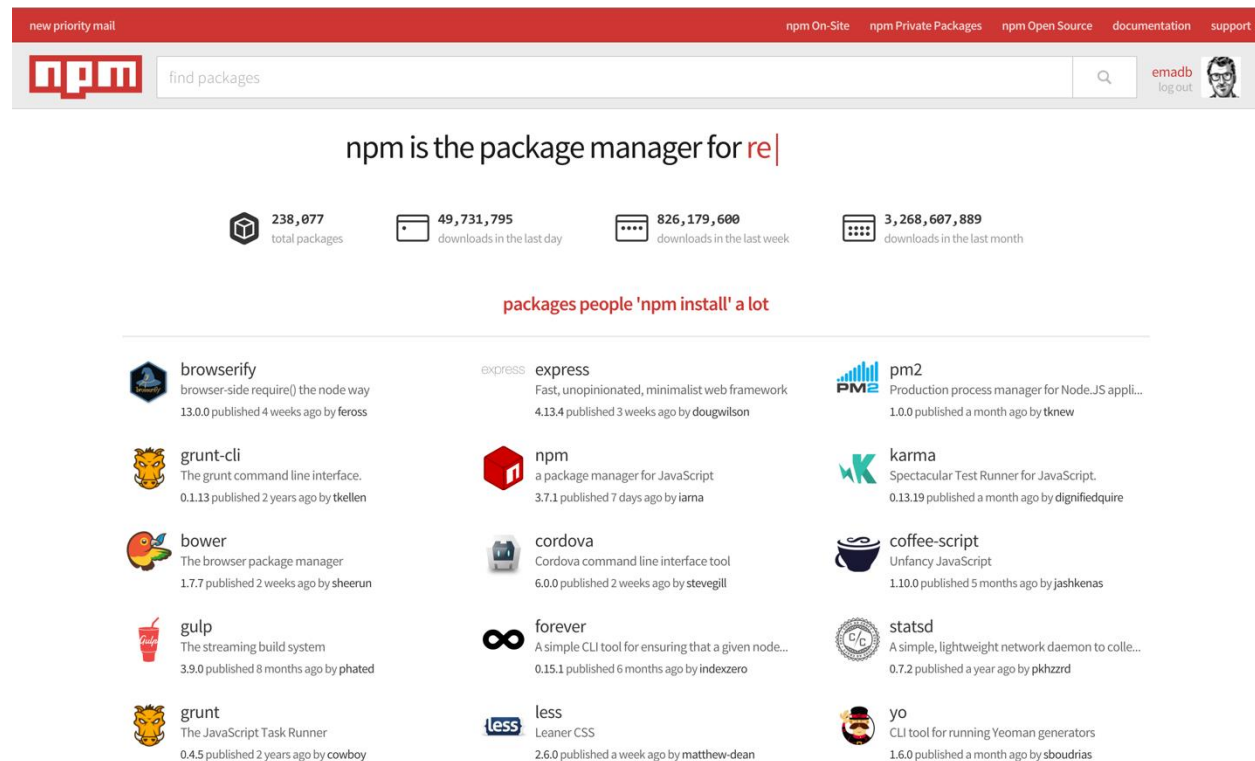


Figure 9: npmjs.com home page

To install a package, we must run this command:

```
> npm install <package name>
```

This command will install the packages in the **node_modules** folder of the current directory and it will be available to other modules in the main directory or in some subfolder of the current directory.

For example, we can try to install Express.js, a framework for writing web applications:

```
> npm install express
```

This command will download the Express package and all its dependencies in the **node_modules** folder so that they will be available for the scripts in the current folder or any nested folders.

npm is also useful for creating new projects. Just run:

```
> npm init
```

This command will run through a simple wizard and create a file named `package.json` with the collected information. Something like:

```
{
  "name": "sampleApp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Emanuele DelBono",
  "license": "ISC"
}
```

This file acts as a sort of project file with some metadata information. Most of the elements are self-explanatory; others will be discussed later.

This file is primarily used for storing dependency information. For example, if we run **npm install express** in the same folder with the **--save** flag, the dependency information will be stored in the `package.json` file:

```
> npm install express --save
```

When this command is completed, the `package.json` file will be updated with this new information:

```
"dependencies": {
  "express": "^4.13.4"
}
```

That informs us that `express 4.13.4` (using semantic versioning) is a dependency of our project. This is useful because usually the `node_modules` folder is not added to source control, and so with `package.json` we can download all the needed dependencies with correct versions just by typing **npm install**.



Tip: It's generally better to stick to a particular version. Sometimes even minor version differences break something. I usually remove the `^` in the `package.json` dependencies section.

The dependencies in the package.json file are usually split into two groups: **dependencies** and **devDependencies**. It is a good practice to put the runtime dependencies inside the **dependencies** group and keep the **devDependencies** section for all the packages that are used only during development (for example, for testing, linting code, seeding the database, etc.). This helps the deployment because in the production environment only the dependencies will be downloaded, leaving the server free of unneeded packages.

In addition to dependency management, the package.json file is useful for other things. It's becoming quite common to use the package.json file to automate some project tasks using the script section.

By default, the script section has one entry dedicated to testing and the default implementation simply echoes a string to the terminal, saying that no test has been configured.

It can be useful to add a run script to start the application. For example, if the entry point is in the index.js file, we can define a start script like this:

```
"scripts": {  
  "start": "node ./index.js"  
},
```

When we execute the command:

```
> npm run start
```

Node.js will start the index.js file as if we are running directly from the terminal. Some hosting services use this script to execute custom commands during the deployment of the application. For example, Heroku runs the post-install script after having installed all the dependencies. It's useful, for example, to precompile the static assets.

In cases where we are developing a library that will be published on npm, the **main** attribute is really important. It tells Node.js which file to include when the library is required.

Summary

npm and package.json are two of the pillars of any Node.js application and it is important to use them correctly, especially when we are working inside a team, so that everybody will be able to work at their best.

Chapter 4 Using the Filesystem and Streams

From this chapter we start writing something interesting, using the filesystem, and discovering the power of the stream.

The fs module

All you need to work with the filesystem is inside the **fs** module that is part of the core library that came with Node.js.

The **fs** module is a sort of wrapper around the standard POSIX functions and exposes a lot of methods to access and work with the filesystem. Most of them are in two flavors: the default async version and the sync version. Even if the async version is preferable, sometimes the sync version is useful for some tasks that don't need extreme performance.

Reading a file

To read a file from the filesystem, we can use the **readFile** function:

Code Listing 22

```
const fs = require('fs')
fs.readFile('/path/to/file', (err, data) => {
  // do something with data
})
```

This is the asynchronous version that calls the callback when the file is read.

The synchronous version is:

Code Listing 23

```
const fs = require('fs')
const data = fs.readFileSync('/path/to/file')
```

The asynchronous version is usually preferable, but there are cases in which you need to wait for the file to be ready before going on.

Writing a file

The counterpart of reading a file is writing it using the **writeFile** function:

Code Listing 24

```
const fs = require('fs')
fs.writeFile('/path/to/file', data, (err) => {
  // check error
})
```

The **writeFile** function has its own synchronous version that works the same called **writeFileSync**.

Watching files

Often applications should monitor a folder and do something when a new file arrives or when something changes. Node.js has a built-in object to watch a folder that emits events when something happens.

Code Listing 25

```
const fs = require('fs')
const watcher = fs.watch('/path/to/folder')
watcher.on('change', function(event, filename) {
  console.log(`${event} on file ${filename}`)
})
```

If we execute this piece of code and try to add a file or change a file in the **/path/to/folder** folder, the **change** event will be executed and the **event** type and **filename** will be printed to the console.

The path module

Another useful module tied to the filesystem is the module **path** that exposes a set of functions that operate with file paths. This module is interesting for managing paths without worrying about separators, concatenation, or file extensions.

For example, if we want to join a path with a file name to get the full path, we can use the **join** method:

Code Listing 26

```
const path = require('path')
const fullPath = path.join('/path/to/folder', 'README.md')
```

fullPath will be **/path/to/folder/README.md**.

Sometimes it's useful to get the full path of a file in the current directory. In these cases, we can use the `__dirname` global variable to build the full path.

Code Listing 27

```
const path = require('path')
const fullPath = path.join(__dirname, 'README.md')
```

This will return the full path of the **README.md** contained in the current folder.

The **path** module also has a method for decomposing a full path in parts:

Code Listing 28

```
const path = require('path')
const parts = path.parse('/path/to/a/file.txt')
console.log(parts)
```

This will print out the parts of the path in the following format:

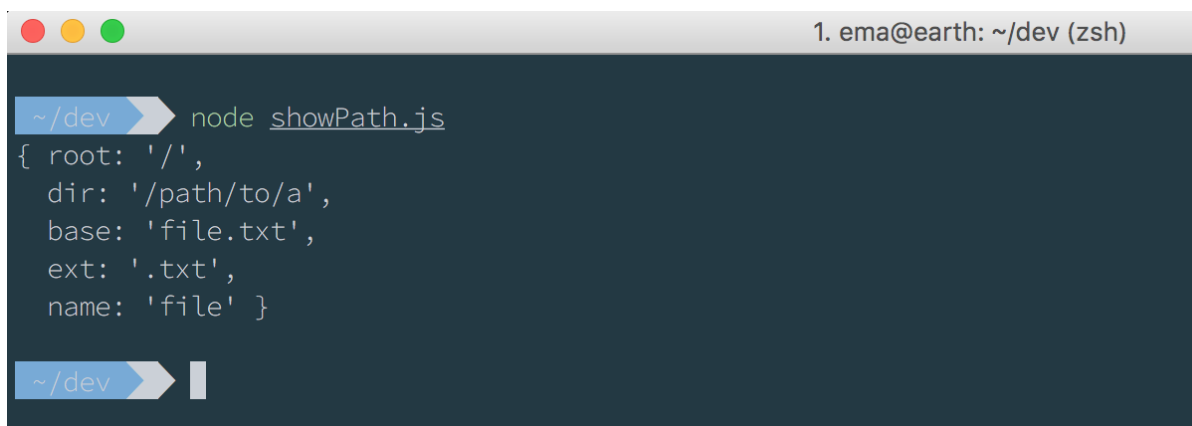


Figure 10: Path info

The **parse** method will read a string and split it into its composing parts.

The **path** module has other methods that you can read about at nodejs.org/api/path.html.

Streams

In [Chapter 2](#), we wrote a sample HTTP server that serves a file from the filesystem. The code was:

Code Listing 29

```
const http = require('http')
const fs = require('fs')
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'});
  fs.readFile('./index.html', (err, file) => {
    response.end(file);
  })
})
server.listen(8000)
```

Like we said in Chapter 2, this code is very harmful because it reads all the file content in memory, and only after having read it all does it respond to the client.

Streams are the perfect solution for these contexts. Streams are similar to UNIX pipes and can be readable, writable, or both (duplex). **The nice thing is that they don't need to read all the data before making it available to others.** They emit events when a chunk of data is available so that the consumers can start using it.

Streams are event emitters and, in the case of readable streams, their two main events are **data** and **end**, which are raised when a chunk of data is ready to be used and when the stream has finished, respectively.

Readable streams

Code Listing 30

```
const fs = require('fs');
const http = require('http');
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'});
  var stream = fs.createReadStream('./index.html');
  stream.pipe(response);
});
server.listen(8000);
```

This small sample introduces streams for reading a file and serving it to the client. It imports the **fs** module and creates a readable stream from the file `index.html`.

Since the response object is a stream, we can pipe the readable stream to **response** so that the server can start serving chunks of `index.html` as soon as they are available.

This basic sample should clarify how streams work. Essentially, we can pipe a readable stream inside a writable stream.

Writable streams

Writable streams are the counterpart of readable ones. They can be created with the function **createWriteStream** and they are streams on which we can pipe something. For example, we can use a writable stream to copy a file:

Code Listing 31

```
const fs = require('fs');
var sourceFile = fs.createReadStream('path/to/source.txt');
var destinationFile = fs.createWriteStream('path/to/dest.txt');

sourceFile.on('data', function(chunk) {
  destinationFile.write(chunk);
});
```

Streams are event emitters and when some data is available inside the streams, they emit the **data** event (like in the previous example). The **data** event receives a chunk of data that can be used.

When the stream is finished, the emitted event is **end** and can be used to close the stream or to do some final operations.

Summary

Streams are very powerful tools to manage big files or images. Most of the time, the libraries that we use will hide the implementation details, but usually, when dealing with big chunks of data, streams are widely used.

Chapter 5 Writing Web Applications

The most common application type written in Node is web applications. We already saw in [Chapter 2](#) how to write a simple HTTP server. In this chapter, we will go deeper and use the [Express.js framework](#) to see how to build real web applications.

The http module

In Chapter 2, we saw how to write a really basic web server:

Code Listing 32

```
const http = require('http')
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'})
  response.end('<h1>Hello from Node.js</h1>')
})
server.listen(8000)
```

This piece of code creates an HTTP server that listens on port 8000. Whatever requests arrive at the server, the response is always **<h1>Hello from Node.js</h1>**. We are using the **end** method here because we have just one string to send to the client. In general, we could use **response.write** several times and only at the end call **response.end()** to close the connection.

The **request** object contains all the information about the request, so we could use it to decide what kind of response we have to send to the client.

The **response** object is used to build the response. We are using it to add the status code (**200**) and the **Content-Type** header.

If we would like to build something less trivial, we can inspect the **request.url** to decide which content to send to the client.

For example, we can write something like this that sends a different response based on the requested URL:

Code Listing 33

```
const http = require('http')
const server = http.createServer((request, response) => {
  response.writeHead(200, {'Content-Type': 'text/html'})
  if (request.url === '/about') {
    response.write('<h1>About Node.js</h1>')
  } else {
```

```
    response.write('<h1>Hello from Node.js/h1>')
  }
  response.end();
})
server.listen(8000)
```

The **request** object has a lot of other attributes and methods to analyze the request. For example, the **attribute** method contains the type of request (GET, POST, PUT, ...).

The **http** module is quite easy to use and, added to the fact that HTTP protocol is also easy, we can think of writing our own web applications using the **http** module directly. But, even if it is easy, the Node community has created a lot of libraries and frameworks that speed up the development of web applications. One of the first and most famous is Express.js.

Express.js

Express.js is a minimal web framework built on the idea of composing middleware (an idea that came from the Ruby world with Rack). Even if Express is very minimal, the Node community has built tons of plugins to add functionality to Express so that we can choose exactly what we need in our applications.

The minimal Express application is something like this:

Code Listing 34

```
const express = require('express')
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
});

app.listen(8000, () => {
  console.log('Example app listening on port 8000!');
});
```

Before running this code, we need to install Express in our **node_modules** folder using **npm**:

Code Listing 35

```
> npm install express
```

The code requires the **express** module, a function that is used to create an application. With the application instance, we can define the routes and the callbacks to use when the routes have been called.

In the previous example, we have defined a single route that responds to an HTTP GET / (root) and we respond with the string **Hello World!**

This example is not very different from the previous one that used the raw **http** module, but the structure is a little bit different and more maintainable and extensible.

From here we can add more routes using the HTTP methods **get**, **post**, **put**, etc., and specifying the pattern that the route must match.

For example, a basic CRUD API could be something like this:

Code Listing 36

```
const express = require('express')
const app = express()

app.get('/users', (req, res) => {
  // get all users
})

app.get('/users/:id', (req, res) => {
  // get the user with the specified id
})

app.post('/users', (req, res) => {
  // create a new user
})

app.put('/users/:id', (req, res) => {
  // update the user with specified id
})

app.delete('/users/:id', (req, res) => {
  // delete the user with specified id
})

app.listen(8000, () => {
  console.log('Example app listening on port 8000!');
})
```



Note: *CRUD stands for Create, Read, Update, and Delete. They are the four basic operations on a database or, more generally, on a resource.*

On an **app** object, we attach the various routes specifying the method used. This API works on hypothetical user resources and exposes a set of endpoints to manipulate users. Notice the **get** and **post** that access a single user. In these cases, the URL will contain the **id** of the user (something like **/users/42**), and in the route declaration Express uses the **:id** (colon id) to specify that that part is variable. Inside the request we will find the value of **id** inside **req.params.id**.

To build the API responses, we can use the **res** object that is available inside the callback. For example, to return a list of users, we can simply send the user array to the client using the method **send**:

Code Listing 37

```
const express = require('express')
const app = express()

app.get('/users', (req, res) => {
  const users = [{id: 1, name: 'Emanuele'}, {id: 2, name: 'Tessa'}]
  res.send(users)
})
app.listen(8000, () => {
  console.log('Example app listening on port 8000!');
})
```

Express.js treats the **users** array as an object, serializes it as a JSON array (by default), and sets all the necessary headers and status codes to compose the correct response.

If we call the API using **curl**:

Code Listing 38

```
curl -i http://localhost:8000/users
```

The response is:

Code Listing 39

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 52
ETag: W/"34-TWdsT6/Z03egRrB46eWd+w"
Date: Tue, 23 Feb 2016 16:23:37 GMT
Connection: keep-alive

[{"id":1,"name":"emanuele"}, {"id":2,"name":"Tessa"}]
```



Tip: Even if **res.send** converts the object in JSON format, response has a JSON method that explicitly specifies the content type to *application/json*.

Sometimes we need to configure the response, for example, to specify a particular status code or a specific header. The **res** object has a series of methods to do this.

If we need to specify a particular status code, we can use **res.status(code)** where **code** is the actual status code that we want. By default, it's 200.

The **send** and **json** methods have an overload to specify the status code:

Code Listing 40

```
res.json(200, users)
res.send(200, users)
```

To add a header, we use the method **res.set(headerName, headerValue)**.

For some headers there exists an ad hoc method. For example, it is good practice for an API that creates a resource to return the URL of the newly created resource in the **location** header:

Code Listing 41

```
app.post('/users', (req, res) => {
  // create a new user
  const user = createUser(req.body)
  res.location(`/users/${user.id}`)
  res.status(201)
})
```

This example shows a possible implementation of a post to create a new user using the **createUser** function. Then we add the location header and we send the status code 201 (created) to the client. In this case, the body is empty.

Until now we have seen how to use Express.js to create a simple API, but Express is not only about APIs, it permits us to create complete web applications with HTML pages using a template engine.

Jade

In case we need to render an HTML page using a server-side template engine, [Jade](#) is one of the many options available for Express.js.

To start using Jade, we first need to install it:

Code Listing 42

```
npm install jade
```

Then we need to set a couple of options on our application object:

Code Listing 43

```
app.set('view engine', 'jade')
app.set('views', './views')
```

The first **set** specifies that the view engine we want to use is Jade. The second tells the app that the view files will be in the **./views** folder of the current project.

After these couple of settings, we are ready to create our first template and to define a router that will render it.

Creating a template is quite easy; Jade has a very minimal syntax and is easy to learn. For example, we can create a new file **hello.jade** in the **./views** folder:

Code Listing 44

```
html
  head
    title= title
  body
    h1= messageTitle
    div(class='container')
      = messageText
```

This is a Jade template that compiles to:

Code Listing 45

```
<html>
  <head>
    <title>Hey</title>
  </head>
  <body>
    <h1>Hello noders</h1>
    <div class="container">
      lorem ipsum...
    </div>
  </body>
</html>
```

The compilation process is managed by Express with Jade in response to a call to the selected route. In our case we can define a route:

Code Listing 46

```
app.get('/hello', function (req, res) {
  res.render('hello', {
    title: 'Hey',
    messageTitle: 'Hello noders',
    messageText: 'lorem ipsum....'
  })
})
```

This route responds to `/hello` and it renders a view called `hello` (that matches the file name). The `render` method receives the object that we want to pass to the view. This object is composed of three properties: `title`, `messageTitle`, and `messageText`.

So calling the route `/hello` from the browser will render an HTML page like this:

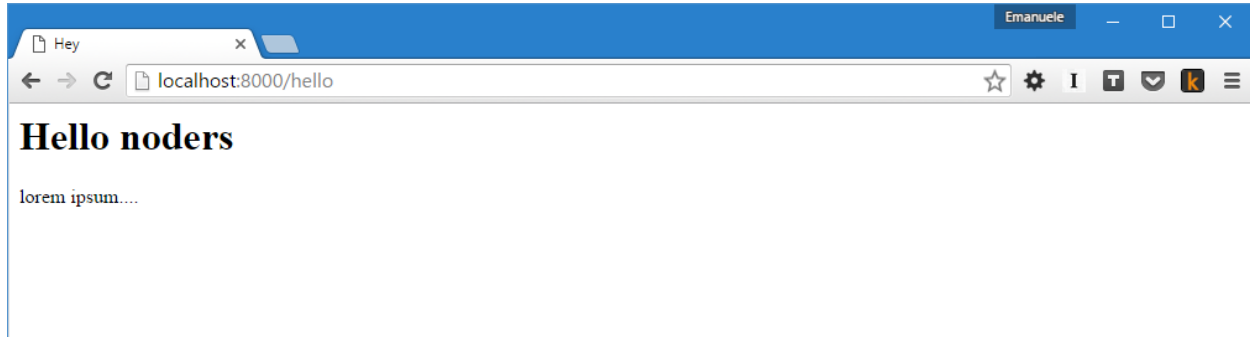


Figure 11: Hello world in Express

Jade is a full template language that supports all the semantics needed to build an HTML page. We can use conditionals, server-side code, and variables. The nice thing is that you don't need to close the tags, so the Jade files are quite compact.

Middleware

As stated previously, Express is a minimal framework and is based on the idea of middleware. Middleware is a stack of functions which an object's request and response pass through. Each middleware can do whatever it needs to the request and response and pass it to the next middleware. It is a sort of chain of responsibility.



Note: Chain of responsibility is a design pattern that aims to create a series of operations on an object. Each element of the chain does something to the input object and passes the result to the next element of the chain.

With this powerful idea, we can compose the stack with the exact middleware that we need. For example, if we need to manage a cookie to identify the user, we can use a middleware that checks the cookie, and if it is not valid, then we can break the chain with a short circuit that returns an HTTP 401 status to the client.

Writing middleware is very easy; it is just a function with three parameters: the request, the response, and the next middleware to call:

Code Listing 47

```
function myMiddleware(req, res, next) {  
  // do what you need and if everything is ok  
  next()  
}
```

```
}  
app.use(myMiddleware)
```

What can we do with middleware?

Middleware functions can execute some code, or make changes to the request or response objects. At the end, they can end the request-response cycle or call the next function to continue the chain of middleware.

With middleware we can, for example, write a logger:

Code Listing 48

```
function requestLogger(req, res, next) {  
  console.log(`${req.method} ${req.originalUrl}`)  
  next()  
}  
app.use(requestLogger)
```

This simple middleware function logs to the console the method and the URL requested.

The idea of function middleware is very powerful and its ease of use is one of the reasons for the success of Express.js.

Most Express.js plugins are implemented as plugins and we can find a lot of middleware for every application need, from security to logging, on npmjs.com.

Middleware can be set for the entire application using the **app.use** function, or per route passing the function directly on the route definition:

Code Listing 49

```
function checkPermissionMiddleware(req, res, next){  
  // verify permissions. If ok call next()  
  if (ok){  
    next()  
  } else {  
    res.status(401)  
  }  
}  
  
app.post('/users', checkPermissionMiddleware, (req, res) => {  
  // create a new user  
  const user = createUser(req.body)  
  res.location(`/users/${user.id}`)  
  res.status(201)  
}))
```

With this configuration, the **checkPermissionMiddleware** function will be called before every **post** request to the route **/users**. It can check to see if the correct authorization headers are in the **req** object to manage continuing the request. If they are present, it will call the next middleware (actually the function defined on the **/users** route), otherwise it will return a 401 without even calling the subsequent middleware.

The ease of modularity of middleware gives us a lot of flexibility with strengthened security and without impacting the performance. In this last example, if the user does not have permission to access the resource, the dangerous code is not even executed.

Chapter 6 Real-Time Apps with WebSocket

Web applications have evolved quickly in recent years, powered by the client-side frameworks that have helped to create real responsive and reactive user interfaces that are indistinguishable from old GUI applications.

If the clients are ready for fast reactivity from the user, we need reactive servers to build a real reactive application too, and that's where WebSocket comes in.

WebSocket

The protocol at the base of every web application is HTTP, and in general HTTP works on a request/response paradigm: the client issues a request to the server and waits for a response. The server never chooses to call the clients. It is passive and just waits for requests.

WebSocket breaks this rule by giving the server the ability to call the clients. It is a standard (IETF as [RFC 6455](#)) and is supported by all modern browsers (starting from IE10).

What WebSocket does is open a full-duplex connection over a TCP connection using port 80, and after a quick handshake between the client and the server, the server keeps the connection open.

Various libraries are available for implementing a WebSocket server. One of the most used is Socket.IO.

Socket.IO

Socket.IO is one of the libraries that implements the WebSocket protocol for Node.js. It is composed of two parts: the client side that runs in the browser and the server side that runs on the server.

The server side is the one that opens a listener for the incoming connections and waits for calls from the clients.

A typical basic setup for Socket.IO in the context of an Express.js application is:

Code Listing 50

```
const app = require('express')()
const http = require('http').server(app)
const io = require('socket.io')(http)

io.on('connection', socket => {
  console.log('a new user is connected');
})
```

```
});  
  
http.listen(3000, function(){  
  console.log('listening on *:3000');  
});
```

To run this code, we should install the Socket.IO package using **npm**.

This simple snippet creates an instance of Socket.IO based on the **http** module of Node.js. This instance is then attached to the **connection** event and every time a user initializes a new connection, the message **a new user is connected** will be printed to the console.

To test this program, we must have a client-side script that initializes the connection. The easiest way to do this is to create an HTML page served by Express.js and put the JavaScript code there.

To do this, we can modify our server-side code:

Code Listing 51

```
const app = require('express')()  
const http = require('http').server(app)  
const io = require('socket.io')(http)  
  
io.on('connection', socket => {  
  console.log('a new user is connected');  
});  
  
app.get('/', (req, res) => {  
  res.sendFile('index.html');  
});  
  
http.listen(3000, () => {  
  console.log('listening on *:3000');  
});
```

We added a **get** route that serves a static file, `index.html`. Here, for the sake of simplicity, we are not using a template engine but a plain HTML file. In more complex cases, we can use the Jade engine, as we saw in the [previous chapter](#).

The HTML will be something like this:

Code Listing 52

```
<!DOCTYPE html>  
<html>  
  <head>
```



```

    <meta charset="utf-8">
    <script src="/socket.io/socket.io.js"></script>
  </head>
  <body>
    <script>
      var socket = io.connect('http://localhost:3000');
    </script>
  </body>
</html>

```

In the HTML page, we require the **socket.io.js** client library that creates a global **io** function. Calling that function and passing the server URL, we establish a new connection with the server.

Now the client and server are connected and one can call the other. For example, if the client needs to send a message to the server, it just has to call the **emit** function:

Code Listing 53

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="/socket.io/socket.io.js"></script>
  </head>
  <body>
    <script>
      var socket = io.connect('http://localhost:3000')
      socket.emit('hello-message', {myData: 'hello from client'})
    </script>
  </body>
</html>

```

The **emit** function takes two parameters, the event name that is a sort of key identifier of the event, and the object that will be passed to the server.

To receive the event, the server must listen for that:

Code Listing 54

```

const app = require('express')()
const http = require('http').Server(app)
const io = require('socket.io')(http)

io.on('connection', socket => {
  socket.on('hello-message', data => console.log(data))
});

```

```

app.get('/', (req, res) => {
  res.sendFile('index.html');
});

http.listen(3000, () => {
  console.log('listening on *:3000');
});

```

After the connection has been established, we can add a handler for the **hello-message** event and every time it arrives from the client, it will print the data to the console.

But the nice thing about WebSocket is that the connection is bidirectional and the server can call the client. To do this, we use the same pattern we just used for the communication from client to server. The server will emit an event and every connected client will receive it:

Code Listing 55

```

const app = require('express')()
const http = require('http').server(app)
const io = require('socket.io')(http)

io.on('connection', (socket) => {
  socket.on('hello-message', data => console.log(data))
  socket.emit('hello-server', {message: 'hello clients'})
});

app.get('/', (req, res) => {
  res.sendFile('index.html');
});

http.listen(3000, () => {
  console.log('listening on *:3000');
});

```

And the client:

Code Listing 56

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="/socket.io/socket.io.js"></script>
  </head>
  <body>
    <script>
      var socket = io.connect('http://localhost:3000')
    </script>
  </body>
</html>

```

```
    socket.emit('hello-message', {myData: 'hello from client'})  
    socket.on('hello-server', function(data) { console.log(data) })  
  </script>  
</body>  
</html>
```

Chapter 7 Accessing the Database

Every application sooner or later needs storage to persist its data. Node.js has several packages to communicate with almost every database. In this chapter, we will see some basic operations with [PostgreSQL](#) and [MongoDB](#), two of the most used databases in their own categories.

Accessing PostgreSQL

PostgreSQL is a relational open source database used in virtually every context. To use PostgreSQL with Node.js, there are several packages available, but the most used is **pg**, the basic driver.

After installing the driver using **npm**, we can connect to a database using this code:

Code Listing 57

```
const pg = require('pg')
const conString = "postgres://usr:pwd@serverName/databaseName"

pg.connect(conString, (err, client) => {
  // Do something with client here
})
```

The syntax is very simple and easy to understand. Now that we have an instance of the client connection to the database, we can issue a query to extract some data.

For the following examples, we will consider a table like this:

Table 2: Sample data

Sample Table: movies		
id	title	year
1	Her	2013
2	I, Robot	2004
3	A.I. Artificial Intelligence	2001
4	2001: A Space Odyssey	1968
5	Blade Runner	1982

Code Listing 58

```
const pg = require('pg')
const conString = "postgres://usr:pwd@serverName/databaseName"

pg.connect(conString, (err, client) => {
  client.query("SELECT * FROM movies", (err, res) => {
    // res.rows contains the record array
  })
})
```

This code runs a query on movies that extracts all the records inside the table. The records are available inside the **rows** property of the result object and it is an array that we can use to access the data.

If we need to specify a particular parameter, we can run a parameterized query. For example, suppose that we need to extract all the movies that were released in 1982. We just add a **where** clause with a parameter:

Code Listing 59

```
const pg = require('pg')
const conString = "postgres://usr:pwd@serverName/databaseName"
const year = 1982

pg.connect(conString, (err, client) => {
  client.query({text:"SELECT * FROM movies WHERE year = $1", values:
    [year]}}, (err, res) => {
    // do something with data
  })
})
```

In this case, we pass to **query** an object with a **text** property (the query text) where the placeholder **\$1** marks the position of the first parameter. The values of the parameters are in the **values** array (ordered) and the driver will check for dirty strings to avoid SQL injection.

The **node-pg** driver supports an event-based API for consuming the result of a **SELECT** statement. The **client** object has two events: **row** and **end**. The **row** event is raised on every new row extracted from the database while the **end** event arrives when no more rows are available:

Code Listing 60

```
const pg = require('pg')
const conString = "postgres://usr:pwd@serverName/databaseName"
const movies = []
const query = client.query('select title, year from movies')
query.on('row', function(row, result) {
  movies.push(row)
})
```

```
});
query.on('end', function(result) {
  console.log(movies.length + ' movies were load');
});
```

The use of the **query** object is a little bit different here: with the **client.query** function we obtain an object to which we can attach some events. The **query** object is basically an event emitter.

On the **row** event, we simply collect the row into an array, and on the **end** event, we print the message to the console.

Which API we should use depends on the context in which we need to use the data. The first is more suitable in cases where you need all the data before going on to something else. The latter is based on events and is preferable where you can return partial results and start using them as soon as they arrive.

So querying the database is quite easy. But what about other CRUD operations?

The pattern is the same: we need to create a query (INSERT, UPDATE, or DELETE) and pass the parameters with correct values.

For example, suppose that we want to add a new movie to our list:

Code Listing 61

```
const pg = require('pg')
const conString = "postgres://usr:pwd@serverName/databaseName"
const year = 1982

pg.connect(conString, (err, client) => {
  client.query({text:"INSERT INTO movies(title, year) VALUES($1, $2)",
    values: ['The Matrix','1999']}}, (err, res) => {
    // check err to see if everything is ok.
  })
})
```

For delete and update, the syntax is exactly the same.

These are some uses of the Postgres driver; more advanced use cases are available on the driver's [GitHub page](#).

In cases where we need more than just a thin driver over the database for Postgres, there are a bunch of other options available as plugins of this driver or as alternatives.

One different approach is to use a real Object Relation Mapper (ORM) to map our own objects to the tables.

The most used ORM in Node is [Sequelize](#). Sequelize is not only for Postgres; it supports different databases like MySQL, MariaDB, and SQLite too.

With Sequelize, we can define an object that will be stored inside a table:

Code Listing 62

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('databaseName', usr, 'pwd');

var Movie = sequelize.define('movie', {
  title: Sequelize.STRING,
  year: Sequelize.INTEGER
})

Movie
  .findAll({where: ['year = ?', 1982]})
  .success(movies => {
    // do something with movie
  })
```

These few lines of code create a **Movie** schema and query the database for all the movies of 1982. As you can see, the query is written using a domain-specific language (DSL) and not SQL. The mapping (which table and which fields) is defined in the schema using **sequelize.define**.

The **Movie** object can also be used for storing new movies to the database:

Code Listing 63

```
const Sequelize = require('sequelize')
const sequelize = new Sequelize('databaseName', usr, 'pwd')

const Movie = sequelize.define('movie', {
  title: Sequelize.STRING,
  year: Sequelize.INTEGER
})

const movie = Movie.build({title: 'The Matrix',year: 1999})
movie.save()
```

The **build** method creates a new movie in memory that can be persisted using the **Save** method. As in the **findAll** example, we don't need to specify the table and the fields in which we store the object properties. All of that is done automatically thanks to the mapping.

The ORM simplifies a lot of the basic operations, but as with every dependency in our projects, it creates a sort of friction. Sequelize supports a lot of features, such as relations that permit you to load related tables starting from one object, but in some ways we lose control of the access to the database, which, in big applications, can lead to performance problems.

Accessing MongoDB

[MongoDB](#) is a different type of database, classified as a NoSQL database, and is more of a document database since the minimal granularity of information that you can store is a document.



Note: NoSQL stands for “Not Only SQL” and represents a database that uses a different mechanism besides relational to store data. There are many options. MongoDB stores records as JSON documents, but others use different storage techniques. Wikipedia provides a good explanation of various database types at en.wikipedia.org/wiki/NoSQL.

A document in MongoDB is JSON (specifically BSON, binary JSON) that we can save inside a collection (a sort of table).

As with Postgres, there are various packages available to access the database, but the MongoDB team has released an official driver called **mongodb**.

The basic template to start using the MongoDB database is:

Code Listing 64

```
const client = require('mongodb').MongoClient

const url = 'mongodb://localhost:27017/mydatabase'
client.connect(url, (err, db) => {
  // do something here
  db.close()
});
```

This code is very similar to what we wrote to connect to a Postgres database. We use a connection string to identify the database server and we call the **connect** function to connect.

Querying a collection to read the documents is also very similar:

Code Listing 65

```
const client = require('mongodb').MongoClient

const url = 'mongodb://localhost:27017/mydatabase'
client.connect(url, function(err, db) {
  const collection = db.collection('movies')
  collection.find({year: 1982}).toArray((err, movies) => {
    // do something with movies array
  })
  db.close();
});
```


Even if this code is very similar to the Postgres code, here we have an additional step to transform the result into a JavaScript array. By default, the **mongodb** driver returns a cursor that we can use to cycle through the elements, or like in the previous example, transform the cursor into a standard array to use the data.

The query language used by MongoDB to extract data is a sort of query by example. In the example, we are asking to extract all the documents that have the property **year** equal to 1982.

To extract all the documents, without any clause, we can pass to **find** an empty object:

Code Listing 66

```
// ...
collection.find({}).toArray((err, movies) => {
  // do something with movies array
})
```

To better grasp the peculiarities of the MongoDB query syntax, the [official documentation](#) is the best resource.

The other CRUD operations are very easy. Remember that in MongoDB you always operate on a document, so we can insert a document, modify a document, and delete a document.

Code Listing 67

```
const client = require('mongodb').MongoClient
const movie = {title: 'The Matrix', year: 1999}
const url = 'mongodb://localhost:27017/mydatabase'
client.connect(url, function(err, db) {
  const collection = db.collection('movies')
  collection.insertOne(movie, (err, r) => {
    // do something with movies array
  })
  db.close();
});
```

The nice thing about using MongoDB with Node is that they speak the same language. We can transparently pass plain JavaScript objects to the driver and it will persist them without needing to transform them. MongoDB uses JSON as its storage format, and Node understands JSON!

The **insertOne** command persists the object inside MongoDB. Inside the callback, the object **r** is an **insertOneWriteOpResult**, a sort of result of the **insert** operation. It contains two interesting attributes: **insertedCount**, which should be equal to one since we have inserted just one document, and **insertedId**, the identifier that MongoDB has assigned to our document.

If we don't specify an **_id** attribute, MongoDB assigns a value of type **ObjectId** (a sort of UUID generated by MongoDB). This is the **id** of the document and we can use it to query the document.

The **mongodb** driver is very low level and the API is sometimes tedious to use. One of the most interesting packages that works with MongoDB is [Mongoose.js](#), an object data mapper (ODM) for MongoDB.

Mongoose.js provides some facilities to work with MongoDB. It permits us to define a schema for our collection and work with objects in an active record style, providing validation, pipeline interception, and a query language.

Mongoose.js needs a couple of things to work: a schema and an instance of the model. To create a schema, we have to define an object using the Mongoose syntax. For example, for our movie database:

Code Listing 68

```
// movieSchema.js
const mongoose = require('mongoose')
const movie = mongoose.Schema({
  title: String,
  year: Number
})
module.exports = mongoose.model('Movie', movie)
```

Using the function **Schema**, we define a schema for our collection, in this case a couple of properties with their own type.



Note: Even if MongoDB is a schemaless database, it's very rare to store in a collection objects with different schemata. In cases like this, you can use a plugin called *mongoose-schema-extend*.

At the end of the module, we create a model based on that schema. Now we can use this model to query the database:

Code Listing 69

```
const mongoose = require('mongoose')
const Movie = require('./movieSchema')

Movie.find({year: 1989}, (err, movies) => {
  // do something with movies array
})
```

By requiring the **Movie** model, we can use it to query the database. The query syntax is like the previous one with the default driver, but in this case we don't have to transform the cursor into an array since what we have from Mongoose is already an array of documents.

The real power of Mongoose is in the other operations. For example, creating a new movie and saving it to the database is very easy:

Code Listing 70

```
const mongoose = require('mongoose')
const Movie = require('./movieSchema')

const movie = new Movie({title:'The Matrix', year: 1999})
movie.save()
```

In this example, we have created a new instance of a movie specifying the title and year. Having a Mongoose model, we can just call **save** to persist it to the database. The **save** function receives an optional callback with the error, so the preceding example is a simplified version that does not check for errors.

Mongoose manages many other things that are beyond the scope of this book. [The project's website](#) is well documented and full of useful examples to follow.

So persisting data with Node.js is not so different from other platforms. MongoDB is the database with less friction since it works with JSON data, but not all applications need a document database, and sometimes the good old SQL database is the safer choice.

Chapter 8 Messaging with RabbitMQ

Until now we have seen monolithic applications, but sometimes, as we will see in the next chapter, splitting our application into multiple parts can be a good way to make them more maintainable. But as soon as we have two different applications that share some responsibility, they need to communicate.

[RabbitMQ](#) is a message broker that implements the [Advanced Message Queuing Protocol](#) (AMQP). In practice, RabbitMQ is a queue system that manages the queueing and delivery of messages.

The basic concepts of RabbitMQ to know before starting are the producers, consumers, exchanges, and queues.

- Producers create new messages and send them to a RabbitMQ server.
- Consumers wait for messages and, when one is ready, they receive them.
- Exchanges receive messages from the producers and push them to one or more queues.
- Queues are where messages are stored before a consumer consumes them. It is a sort of mailbox that contains messages delivered from exchanges.

The exchanges know which queue a message must be sent to because of the binding between the exchange and the queue.

So with RabbitMQ, one of our applications can publish a message on a queue, while another application can listen for messages and dequeue them from the queue.

As for databases, there are several packages to interact with RabbitMQ. The most used in this context is `amqpjs`, so to start using RabbitMQ, we must install this package.



Note: *RabbitMQ must be installed on your computer. Instructions can be found on the [official website](#) and differ depending on your operating system.*

RabbitMQ is big and complex. It supports a lot of patterns and messaging architectures. In this chapter we will see the classic publish–subscribe pattern.

The `amqpjs` library is very low level, but its advantage is that it permits us to greatly customize the interaction between our application and RabbitMQ.

The classic publisher is something like this:

Code Listing 71

```
const amqp = require('amqplib/callback_api')

amqp.connect('amqp://localhost', (err, conn) => {
  conn.createChannel((err, ch) => {
    ch.assertExchange('myExchange', 'fanout', {durable: true})
    ch.publish('myExchange', '', new Buffer('A message'))
  })
})
```

This snippet creates a new connection to an instance of RabbitMQ server and on this connection it creates a channel for communication. With the channel, we assert that the exchange **myExchange** exists, and if not, we create it. After that, we simply publish a message to the exchange **myExchange**. A message must be passed as a **Buffer**, so in this example we convert the string **A message** to a buffer using the native Node **Buffer** object. That's all for publishing.

Receiving a message follows the same pattern:

Code Listing 72

```
const amqp = require('amqplib/callback_api')

amqp.connect('amqp://localhost', (err, conn) => {
  conn.createChannel((err, ch) => {
    ch.assertExchange('myExchange', 'fanout', {durable: true})
    ch.assertQueue('myQueue', {durable: true}, (err, q) => {
      ch.bindQueue(q.queue, ex, '')
      ch.consume(q.queue, function(msg) {
        console.log(msg.content.toString())
      })
    })
  })
})
```

This code is a little bit more complicated. After creating the connection and the channel, we assert the existence of the exchange. This is necessary in both the publisher and the consumer because we don't know which one arrives first. In any case, we assure the existence of the exchange. After having created the exchange, we must create the queue using the method **assertQueue**. The **assertQueue** receives the name of the queue and some options (in this case we only make the queue persistent so that if the server restarts, the queue is still there). With the instance of the queue, we connect it to the exchange, creating a binding between **myQueue** and **myExchange**. Finally, we attach the consumer function with the **consume** function. This function receives a callback that will be called once a message arrives in the queue and it will simply print the message content to the console.

Why is RabbitMQ useful?

RabbitMQ is useful because it permits us to create a scalable application that can span multiple machines. Consider, for example, a consumer that has some operations to do, and to complete these operations, it needs about one second. If the publish rate is over one message per second, the only consumer will not be able to keep up with the publisher.

But thanks to the architecture, we can run a second instance of the consumer to double the processing rate.

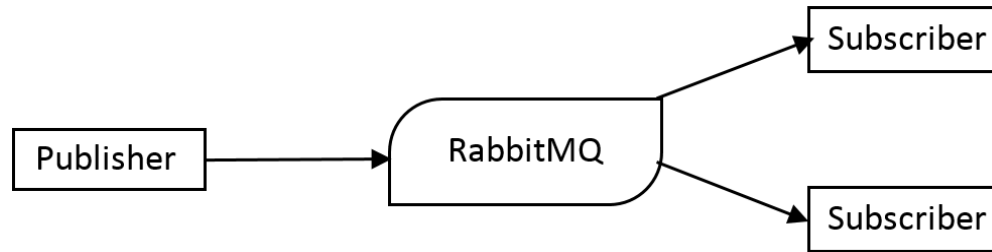


Figure 12: The publisher–subscriber pattern

With this architectural pattern, we keep the doors open to future scalability at an initial cost to setup, and configure an application that is based on messaging.

The publisher–subscriber pattern is just one of the various topologies that we can create with RabbitMQ. RabbitMQ has a concept of exchange that is a piece of the architecture acting as glue between the publisher and the destination queue. Using an exchange, the publisher does not know which queue the message must be sent to; it's the exchange that it is in charge of doing this.

In the publisher–subscriber pattern, the exchange is of type fanout: one message is delivered to different queues. Other exchange types use a routing key to deliver the message to the matching queue. The routing key is a string that identifies or characterizes the message.

The possible exchange types are listed in Table 3.

Table 3: RabbitMQ Exchange Types

Exchange Type	Description
Direct	One message delivered to exactly one queue.
Fanout	One message delivered to different queues (pub–sub pattern).
Topic	Messages are delivered to queues based on content of the routing key (a sort of tag of the message).
Header	Like topic exchange but based on the header content.

Domain-driven design, event sourcing, and CQRS are architectures often used in C# and Java, but not so well known in the Node.js world. But the tools are readily available, and it's time to start thinking big in Node.js, too.

Chapter 9 Support Tools: Build and Testing

Until now we have seen practices and packages that we need to build our applications. But the Node community is also very strong on the side activities of a project, by which I mean testing, building, and everything else that is part of a project.

Mocha and Chai

Testing is one of the most important parts of an application and practices like TDD are widespread in the Node community.

One of the tools that we can use to define and run tests is [Mocha](#). Mocha is a JavaScript test framework that defines a set of functions we can use to write our tests. It comes with a runner and support for asynchronous tests.

Let's say that we have one module like this:

Code Listing 73

```
// simpleMath.js
module.exports = {
  add: (a, b) => a + b,
  sub: (a, b) => a - b,
  multiply: (a, b) => a * b
}
```

Ignore the inutility of this module, it's just something that we can test focusing on the syntax of Mocha.

Let's now write some tests on this module:

Code Listing 74

```
const simpleMath = require('./simpleMath')
const assert = require('chai').assert

describe('Math module', () => {
  it('should add 3 and 4 and return 7', () => {
    const result = simpleMath.add(3, 4)
    assert.equal(7, result);
  });
  it('should subtract 3 from 12 and return 9', () => {
    const result = simpleMath.sub(12, 3)
    assert.equal(9, result);
  });
});
```

```
});
```

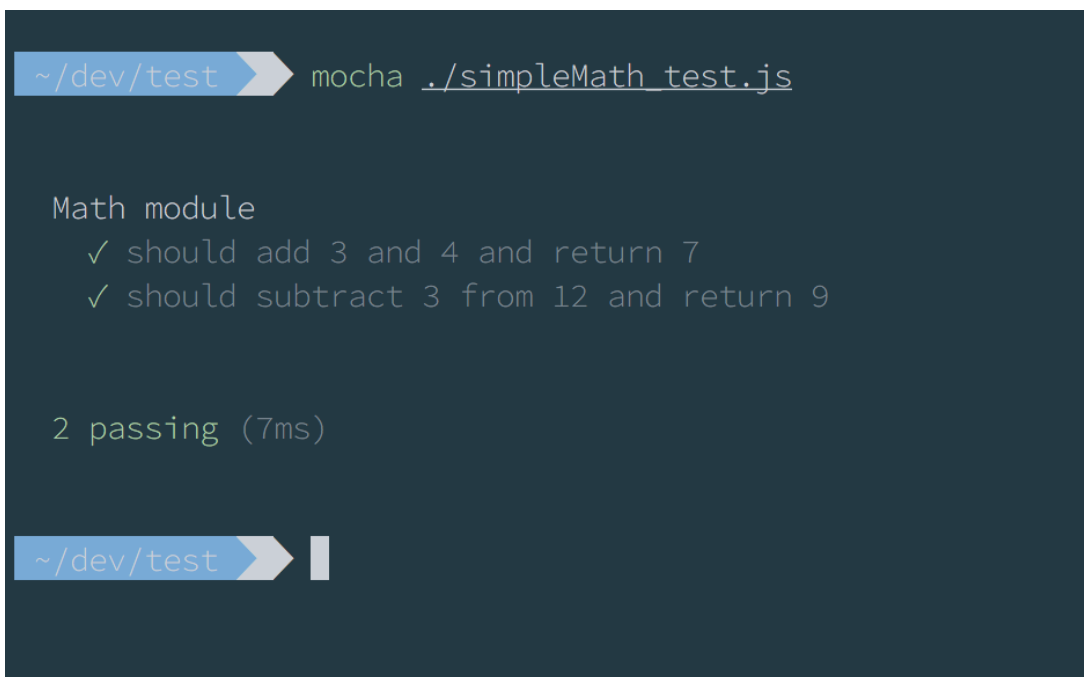
The code is very readable; that is one characteristic of Mocha. It describes the functionality of a module and states that if we add 3 and 4, we must obtain 7 (that is quite obvious).

describe and **it** are two functions of Mocha. We don't need to require Mocha because the tests will be executed by Mocha and that will provide the correct context.

To run the tests from the command line:

Code Listing 75

```
$ mocha ./tests/simpleMath_test.js
```

A terminal window with a dark background. The prompt is ~/dev/test. The command mocha ./simpleMath_test.js has been executed. The output shows 'Math module' followed by two passing tests: '✓ should add 3 and 4 and return 7' and '✓ should subtract 3 from 12 and return 9'. Below these, it says '2 passing (7ms)'. At the bottom, there is another prompt ~/dev/test followed by a vertical bar.

```
~/dev/test ➤ mocha ./simpleMath_test.js

Math module
  ✓ should add 3 and 4 and return 7
  ✓ should subtract 3 from 12 and return 9

2 passing (7ms)

~/dev/test |
```

Figure 13: Test result

Mocha doesn't come with an assert library and leaves developers free to choose what they prefer. In the previous example, we used [Chai](#), which is just an assert library.

But we know that in Node, most code is asynchronous and Mocha supports it out of the box using a **done** callback. Suppose you have this code to test:

Code Listing 76

```
// customerApi.js
module.exports = {
  getCustomers: (callback) => {
    return someApi.get('/customers', (err, res) => {
```



```

        callback(res.body)
      })
    }
  }
}

```

We can think of this as a sort of call to an API that gets a list of customers back. We don't need to go into details. We just need to understand that the final callback will be called when the **get** method returns.

How do we test this?

Since it is asynchronous, we can't just call the **getCustomers** and assert on the callback because the test will end before the callback is called. As mentioned previously, Mocha has support for this context and the solution is to pass a **done** function to the tests and call this function when the test is completed:

Code Listing 77

```

const customerApi = require('./customerApi')
const assert = require('chai').assert

describe('Customer API module', () => {
  it('should return a list of customers', (done) => {
    customerApi.getCustomers(customers => {
      assert.isTrue(customers.length > 0)
      done()
    })
  });
});

```

As we can see, it is very easy to wait for the callback to complete. We assert that the customers are available in the response and after that we call the **done** function to tell Mocha that this test has completed and it can go on to the next one.

Mocha has a default timeout of two seconds. This means that if after two seconds the **done** callback is not called, the test fails.

The last thing to say concerning Mocha is about the support for test hooks. Mocha has some special functions we can implement that can be executed before or after our tests:

Code Listing 78

```

const customerApi = require('./customerApi')
const assert = require('chai').assert

describe('Customer API module', () => {
  before(() => { /* this run once before all tests of this module*/ })
  beforeEach(() => { /* this run before every test of this module*/ })

```

```
it('should run a test', () => {})  
afterEach(() => { /* this run after every test of this module*/ })  
after(() => { /* this run once after all tests of this module*/ })  
});
```

The four methods **before**, **beforeEach**, **after**, and **afterEach** are useful for setup objects and connections, and teardown at the end of the tests.

To run the tests, we must install the **mocha-cli** module as global and run Mocha from the command line, passing the path of the files that contain tests.

Otherwise we can use Gulp to integrate the tests in our build process.

Gulp

If you come from a compiled programming language, builds for JavaScript may sound a bit weird since JavaScript is not compiled as C# or Java are, but the term *build* has a wider meaning and Gulp is one of the tools we can use to "build" our application.

[Gulp](#) is a sort of task runner: it executes a series of tasks as configured. Every task has one objective to complete.

Generally in a Node application, there aren't many tasks to execute and most of them can be done using the **scripts** section of the package.json file. Gulp (and Grunt or similar tools) is used in the front-end world, where the JavaScript must be combined into a single file and minimized.

But Gulp is still an interesting tool, especially in the dev environment, for automating most tasks.

To start with Gulp, we have to install two packages: the **gulp-cli** (with the global option) and Gulp:

Code Listing 79

```
$ npm install gulp-cli -g  
$ npm install gulp --save-dev
```

The **gulp-cli** package creates the symlink to run Gulp from the terminal.

With these two packages installed, we must create a `gulpfile.js` that will contain our tasks. We can start with something like this:

Code Listing 80

```
const gulp = require('gulp')
```

```
gulp.task('default', () => {  
  console.log('gulp! It works')  
})
```

From the command line, we can run Gulp and obtain something like this:

Code Listing 81

```
$ gulp  
[18:02:27] Using gulpfile ~/dev/nodejs_book/gulpfile.js  
[18:02:27] Starting 'default'...  
gulp! It works  
[18:02:27] Finished 'default' after 140 µs
```

As we can see, (apart from the log) it just prints the **gulp! It works**.

The idea behind Gulp is to build a series of tasks that we can run alone or in series using the concepts of streams and pipes. Every task can be an input for the next one to build more complex tasks.

Gulp comes with lots of plugins to help the composition of a useful build pipeline. For example, if we want Gulp to run our Mocha tests, we can install the **gulp-mocha** plugin and configure our `gulpfile.js` like this one:

Code Listing 82

```
const gulp = require('gulp')  
const mocha = require('gulp-mocha')  
  
gulp.task('mocha-tests', () => {  
  return gulp.src('tests/**/*.js')  
    .pipe(mocha({reporter: 'spec', bail: true}))  
    .once('error', (err) => { console.log(err); process.exit(1) })  
    .once('end', () => { process.exit() })  
});  
  
gulp.task('default', ['mocha-tests'])
```

This `gulpfile` defines a new task named **mocha-tests** that reads all the files that match the pattern and pipes the result (the file list) to the **mocha** function that runs the tests. In the case of an error, it prints the error to the console and the process will stop.

Another practical usage of Gulp is to run an application during the development phase and, even more useful, to restart it when something changes.

During the development phase, we usually try the application in the browser, and when we modify some code, we need to restart the application to view the changes.

With Gulp, we can automate this process:

Code Listing 83

```
const gulp = require('gulp')
const nodemon = require('gulp-nodemon')

gulp.task('dev-mode', () => {
  nodemon({script: './app/index.js'})
});

gulp.task('default', ['dev-mode'])
```

We are requiring a new module called **gulp-nodemon**, a Gulp plugin, to enable **nodemon**. The **dev-mode** task starts **nodemon** with an object parameter that specifies the entry point of the application. **nodemon** simply watches the filesystem for changes and when a file changes, it restarts the application. This script is a time-saver during development.

Gulp can automate much more than this and you can find a list on its [website](#). Even if you need time to find the perfect workflow with Gulp, I can say that it is a good investment. Every minute spent in setup will be minutes earned later.

ESLint

We know that JavaScript is not a compiled language and its dynamic nature, even if it is very powerful, is also dangerous if certain rules are not followed.

That's why linters exist.

Linters are tools that analyze the source code and verify that a set of rules are satisfied and, if some rules are broken, they break the build process.

As with everything in Node.js, there are various linters, but lately one of the most used is [ESLint](#) which supports the new ES6 syntax out of the box.

ESLint is an npm package that comes with a binary, but since we've already seen Gulp, we will use it inside Gulp so that linting becomes part of the build process.

So let's start with the gulpfile:

Code Listing 84

```
const gulp = require('gulp')
const eslint = require('gulp-eslint');

gulp.task('eslint', () => {
  return gulp.src(['app/**/*.js', 'tests/**/*.js'])
    .pipe(eslint('eslint.config.json'))
});
```

```

        .pipe(eslint.format())
        .pipe(eslint.failAfterError());
    });

    gulp.task('default', ['eslint'])

```

The structure of the file is the usual of every gulpfile. The task **eslint** is the task that lints our files. It takes a list of files (every .js file in the app folder and in the test folder) and passes it to the ESLint plugin.

The rules to apply are specified in the **eslint.config.json** file that we will see in a minute. After linting the source code, it generates a report on the terminal and if there are errors, it stops the build.

The rule file is a JSON file with the set of rules that we want to enable. ESLint comes with tons of rules, but we are not obliged to use them all. In fact, every team must find the set of rules that fits their conventions:

Code Listing 85

```

{
  "rules": {
    "no-shadow": 2,
    "no-undef-init": 0,
    "no-undef": 2,
    "no-undefined": 0,
    "no-unused-vars": [2, { "vars": "local", "args": "after-used" }],
    "no-use-before-define": 0,
    "arrow-body-style": [0, "as-needed"],
    "arrow-parens": 0,
    "arrow-spacing": [2, { "before": true, "after": true }],
    "comma-dangle": [0, "always-multiline"],
    "constructor-super": 0
  }
}

```

This is part of an example ESLint config file. Every rule has a name and a value:

- **0** if we want to disable the rule.
- **1** if we want the rule to generate a warning.
- **2** if we want the rule to generate an error.

Some rules need some additional options, like arrow-spacing that specifies if we want a space around the arrow of the function in ES6 syntax. We can specify if we want the space before the arrow or after it.

We can find the full list of rules on the [ESLint](#) website and they are all documented with examples.

How do you start with ESLint? Which rules apply? Airbnb has a [repository on GitHub](#) dedicated to JavaScript convention and code style and it contains a good ESLint file that we can use as a starting point. From there we can add, remove, or simply disable rules that are too restrictive or loose.

Appendix A: Introduction to ES6

The JavaScript user community is getting bigger year by year, even if the language is supposedly not one of the better designed or the one with strict rules and solid implementation.

Fortunately, in recent years the ECMA association has been working on improving the language with new constructs and better standardization.

ECMA standardized ES5 in 2009 and in 2015 they standardized ES6, also known as [ECMAScript 2015](#). The feature list of the language is quite long, and in this book we used some of it. In this Appendix, we will briefly go through them to understand what they are.

Node, starting from version 4.4.1, has come to add support for the ES6 features. Today, with version 6.0.0, more than 90% of the ES6 features are implemented in the language.

This means that we can use them actively in our projects.



Note: *If you are already familiar with ES6, you can skip this chapter.*

Arrow functions

Arrow functions are one of the most used features of ES6 for two main reasons: they simplify the code by shortening the function declaration and they automatically bind the parent **this** to the function.

In ES5 we usually write code like this:

Code Listing 86

```
myFunction(42, function(result) {  
  // do something with result  
})
```

With ES6:

Code Listing 87

```
myFunction(42, result => {  
  // do something with result  
})
```

The keyword **function** is no longer necessary. We can use the fat arrow operator to indicate that there is a function with one parameter.

Actually, the two forms are not exactly the same. There is subtle (but useful) difference. Consider these two examples:

Code Listing 88

```
const obj = {
  items: [1,2,3,4],
  offset: 10,
  double: function(){
    return this.items.map(function(i) { return this.offset + (i * 2)
  }.bind(this))
  },
  triple: function() {
    return this.items.map(i => this.offset + (i * 3) )
  }
}
```

As you can see, the **this** is managed differently from one case to another. In the **double** function, we are forced to bind **this** to the **map** callback, otherwise **this.offset** would be undefined and the result of the **map** function would have been an array of **NaN**.

In the **triple** function, we don't need to explicitly set **this** to the **map** function because it uses the parent **this** (**obj**).

Most of the time, the fat arrow implementation is more useful, but not always, so don't forget **this**!

const and let

Up until now, we've been used to using **var** to declare a variable, and we know that **var**-scoped variables are based on the function. Now, with **let**, JavaScript acquires block-scoped variables:

Code Listing 89

```
function myFunction(x) {
  if (x === 42){
    let foo = "I'm a scoped variable"
    // foo is available here
  }
  // foo is not available here
}
```

const is the construct to declare real constants that cannot change value:


```
const x = 9
x = 7 // TypeError: Assignment to constant variable.
```

Template strings

This functionality provides some syntactic sugar to concatenate strings and variables without opening and closing the quotes.

So, for example, a function like this:

Code Listing 91

```
function greet(name){
  return 'hello' + name
}
```

Becomes:

Code Listing 92

```
function greet(name){
  return `hello ${name}`
}
```

It uses the backtick character and `${ }` to express the evaluable part of the string.

Classes

We know that JavaScript is not a class-based language like Java or C#. Inheritance in JavaScript is obtained using the prototype chain. But the ECMAScript committee has decided that **class** is a worthwhile keyword to use even in JavaScript. So, even if there aren't any real classes, we can use the keyword to mimic the creation of an object.

Without classes, we used to do something like this:

Code Listing 93

```
function Person(name, surname){
  this.name = name
  this.surname = surname
}

Person.prototype.getFullName = function(){
```

```
    return this.name + ' ' + this.surname
}
```

Having defined this construction function, we can create instances of **Person** using the new operator:

Code Listing 94

```
var tess = new Person('Tessa', 'Smith')
tess.getFullName() // 'Tessa Smith'
```

With the new keyword, the class definition became something like this:

Code Listing 95

```
class Person {
  constructor(name, surname){
    this.name = name
    this.surname = surname
  }
  getFullName() {
    return this.name + ' ' + this.surname
  }
}
```

For a Java developer, this syntax is surely more clear, but in my opinion, it is misleading since what we are defining is not a real class.

The **class** keyword also supports single inheritance using the **extends** construct:

Code Listing 96

```
class Developer extends Person {
  constructor(name, surname, language){
    super(name, surname)
    this.language = language
  }
  getPreferredLanguage() {
    return language
  }
}
```

Again, the syntax is familiar, but it's not classic inheritance. It is prototypal inheritance, namely a prototype chain.

Destructuring assignment

Destructuring is a new feature of JavaScript that simplifies assignment. It works with arrays and objects in this way:

Code Listing 97

```
var [a,b] = [1,2] // a=1, b=2
var [a,,b] = [1,2,3] // a=1, b=3
var [a,b] = [1,2,3,4,5] // a=1, b=2
```

In the examples, we can see how easy it is to extract values from an array. But destructuring also works with objects:

Code Listing 98

```
var obj = {foo: 42, bar: 'hello'}
var {foo, bar} = obj // foo=42, bar='hello'
```

Here, the name destructuring is quite clear. This construct extracts fields from an object into free-standing variables.

Destructuring is useful with modules that export an object:

Code Listing 99

```
// myModule.js
module.exports = {value:1, fun: function() {...}}

// app.js
var {fun} = require('./mymodule') // fun is the function
```

Default parameters

Until now, function parameters could not have default values. This is the new feature:

Code Listing 100

```
function myFunc(foo, bar = 42){ console.log(foo, bar) }

myFun(10) // 10, 42
myFun(10, 50) // 10, 50
```

Rest and spread

The rest operator is used to collect a set of parameters of a function inside an array. Before ES6, we used to access the arguments variable to read the array of parameters, but now we can do it like this:

Code Listing 101

```
function myFunc(foo, ...bar) {  
  console.log(foo, bar)  
}  
  
myFunc(1) // 1, []  
myFunc(1, 3) // 1, [3]  
myFunc(1,3,5) // 1, [3,5]
```

The spread operator is the inverse:

Code Listing 101

```
function myFunc(foo, bar) {  
  console.log(foo, bar)  
}  
  
var args = [1,2]  
myFunc(...args) // 1, 2
```

Summary

In this Appendix, we examined some of the new constructs and syntax that ECMAScript 2015 brought to JavaScript developers. We focused on the parts that we have used in this book, but there are many more that you can find by reading [the specification](#) or a book dedicated to the new version of JavaScript, such as Syncfusion's [ECMAScript 6 Succinctly](#).