Node.js Design Patterns

Second Edition

Get the best out of Node.js by mastering its most powerful components and patterns to create modular and scalable applications with ease

Mario Casciaro
Luciano Mammino

BIRMINGHAM - MUMBAI

Node.js Design Patterns

Second Edition

Copyright © 2016 Packt Publishing

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

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

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

First published: December 2014

Second edition: July 2016

Production reference: 1110716

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B32PB, UK.

ISBN 978-1-78588-558-7

www.packtpub.com

Credits

Authors

Mario Casciaro

Luciano Mammino

Reviewers

Tane Piper

Joel Purra

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Reshma Raman

Content Development Editor

Onkar Wani

Technical Editor

Prajakta Mhatre

Copy Editor

Safis Editing

Project Coordinator

Ulhas Kambali

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Graphics

Kirk D'Penha

Production Coordinator

Nilesh Mohite

About the Authors

Mario Casciaro is a software engineer and entrepreneur, passionate about technology, science and open source knowledge. Mario graduated with a master's degree in software engineering and started his professional career at IBM where he worked for several years on different enterprise products such as Tivoli Endpoint Manager, Cognos Insight, and SalesConnect. Next, he moved to D4H Technologies, a growing SaaS company, to lead the development of a new bleeding-edge product for managing emergency operations in real time. Currently, Mario is the co-founder and CEO of `Sponsorama.com`, a platform to help online projects raise funds through corporate sponsorship.

Mario is also the author of the first edition of *Node.js Design Patterns*.

Acknowledgments

When I was working on the first edition of this book I would never have thought it would become such a success. My biggest thanks go to all the readers of the first edition of this book, to those who bought it, to those who left a review, and to those who recommended it to their friends on Twitter or on other online forums. And of course, my gratitude also goes to the readers of this second edition; to you who are reading these words, you make all our efforts worthwhile. I also want you to join me in congratulating my friend Luciano, the co-author of this second edition, who did a tremendous job updating and adding new invaluable content to this book. All the merit goes to him as I only had the role of adviser in this second edition. Working on a book is not an easy task, but Luciano impressed me and all the staff at Packt for his dedication, professionalism, and technical skills, demonstrating he can achieve any goal he sets his mind to. It was a pleasure and a honor working with Luciano and I'm looking forward to other great collaborations. I also want to thank all the people who worked on the book, the folks of Packt, the technical reviewers (Tane and Joel) and all the friends who provided valuable suggestions and insights: Anton Whalley (`@dhigit9`), Alessandro Cinelli (`@cirpo`), Andrea Giuliano (`@bit_shark`), and Andrea Mangano (`@ManganoAndrea`). Thanks to all the friends who give me unconditional love, to my family, and most importantly to my girlfriend Miriam, the partner of all my adventures, who brings love and joy in every day of my life. There are still a hundred thousand adventures awaiting us.

Luciano Mammino is a software engineer born in 1987, the same year that the Nintendo released Super Mario Bros in Europe, which by chance is his favorite video-game. He started coding at the age of 12 using his father's old Intel 386, provided only with the DOS operating system and the qBasic interpreter.

After a master's degree in computer science he developed his programming skills mostly as a web developer working mainly as freelancer for companies and startups all around Italy. After a start-up parenthesis of 3 years as CTO and co-founder of `Sbaam.com` in Italy and in Ireland, he decided to relocate in Dublin to work as senior PHP engineer at Smartbox.

He loves developing open source libraries and working with frameworks such as Symfony and Express. He is convinced that the JavaScript fame is still at the very beginning and that this technology will have a huge impact in the future of most of the web-and mobile-related technologies. For this reason, he spends most of his free time improving his knowledge of JavaScript and playing with Node.js.

Acknowledgments

The first huge thanks go to Mario for giving me the opportunity and the trust to work alongside him on the new edition of this book. It was an amazing experience and hopefully just the beginning of a long series of collaborations.

This book was only possible thanks to the incredible and efficient work of the Packt team, especially thanks to the relentless efforts and the patience of Onkar, Reshma, and Prajakta. Also thanks to the reviewers Tane Piper and Joel Purra, their experience with Node.js was crucial to raise the quality of the content provided in this book.

A great hug (and many beers) go to my friends Anton Whalley (`@dhigit9`), Alessandro Cinelli (`@cirpo`), Andrea Giuliano (`@bit_shark`), and Andrea Mangano (`@ManganoAndrea`) for encouraging me all along the way, for sharing with me their experience as developers and for providing meaningful insights on the contents of this book.

Another great thank you goes to Ricardo, Jose, Alberto, Marcin, Nacho, David, Arthur, and all my colleagues at Smartbox for making me love my days at work and for inspiring and motivating me to get better every day as a software engineer. I couldn't ask for a better team.

My deepest gratitude goes to my family, who raised and sustained me in every possible way along my journey. Thanks, mom, for being a constant source of inspiration and strength in my life. Thanks, dad, for all the lessons, the encouragement and the advice, I really miss talking with you, I really miss you. Thanks to my brother Davide and my sister Alessia for being present in the painful and the joyful moments and making me feel part of a great family.

Thanks to Franco and his family for supporting many of my initiatives and for sharing their wisdom and life experience with me.

Kudos to my "nerd" friends Gianluca, Flavio, Antonio, Valerio, and Luca for the great time together and for constantly encouraging me to keep working on this book.

Also kudos to my "less nerdy" friends Damiano, Pietro, and Sebastiano for their friendship and all the laughs and the fun we have when we hang out together in Dublin.

Last, but definitely not least, thanks to my girlfriend Francesca. Thank you for the unconditioned love and for supporting me on every adventure, even the craziest ones. I really look forward to writing the next pages in the book of our life with you.

About the Reviewers

Tane Piper is a full stack developer based in London, UK. For over 10 years He has worked for several agencies and companies delivering software in a variety of languages such as Python, PHP, and JavaScript. He has been working with Node.js since 2010 and was one of the first people talking about server-side JavaScript in the UK and Ireland with several talks in 2011/2012. He was also an early contributor to, and advocate for the jQuery project. Currently he works at a consultancy in London delivering innovative solutions and is mostly writing React and Node applications. Outside of his professional work he is a keen scuba diver and amateur photographer.

I would personally like to thank my girlfriend Elina who has turned my life around in the last two years and encouraged me to take up the task of reviewing this book.

Joel Purra started toying around with computers even before he was in his teens, seeing them as another kind of a video game device. It was not long before he took apart (sometimes broke and subsequently fixed) any computer he came across while playing the latest games on them. It was gaming that led him to discover programming in his early teens when modifying a Lunar Lander game triggered an interest in creating digital tools. Soon after getting an Internet connection at home, he developed his first e-commerce website, and thus his business started; it launched his career at an early age. At the age of 17, Joel started studying computer programming and an energy science program at a nuclear power plant's school. After graduation, he studied to become a second lieutenant telecommunications specialist in the Swedish Army before moving on to study for his master's of science degree in information technology and engineering at Linköping University. He has been involved in start-ups and other companies—both successful and unsuccessful—since 1998, and he has been a consultant since 2007. Born, raised, and educated in Sweden, Joel also enjoys the flexible lifestyle of a freelance developer, having traveled through five continents with his backpack and lived abroad for several years. A learner constantly looking for challenges, one of his goals is to build and evolve software for broad public use. You can visit his website at `http://joelpurra.com/`.

I'd like to thank the open source community for providing the building blocks necessary to compose both small and large software systems even as a freelance consultant. Nanos gigantum humeris insidentes. Remember to commit early, commit often!

www.PacktPub.com

eBooks, discount offers, and more

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

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

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

Why subscribe?

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

Table of Contents

1
Welcome to the Node.js Platform

Some principles and design patterns literally define developer experience with the Node.js platform and its ecosystem; the most peculiar ones are probably its asynchronous nature and its programming style that, in its simplest incarnation, make heavy use of callbacks. It's important that we first dive into these fundamental principles and patterns, not only for writing correct code, but also to be able to take effective design decisions when it comes to solving bigger and more complex problems.

Another aspect that characterizes Node.js is its philosophy. Approaching Node.js is in fact way more than simply learning a new technology; it's also embracing a culture and a community. We will see how this greatly influences the way we design our applications and components, and the way they interact with those created by the community.

In addition to these aspects, it's worth knowing that the latest versions of Node.js introduced support for many of the features described by ES2015 (formerly ES6), which makes the language even more expressive and enjoyable to use. It is important to embrace these new syntactic and functional additions to the language in order to be able to produce more concise and readable code and come up with alternative ways to implement the design patterns that we are going to see throughout this book.

In this chapter, we will learn the following topics:

- The Node.js philosophy, the “Node way”
- Node.js version 6 and ES2015
- The reactor pattern—the mechanism at the heart of the Node.js asynchronous architecture

The Node.js philosophy

Every platform has its own philosophy—a set of principles and guidelines that are generally accepted by the community, an ideology of doing things that influences the evolution of a platform, and how applications are developed and designed. Some of these principles arise from the technology itself, some of them are enabled by its ecosystem, some are just trends in the community, and others are evolutions of different ideologies. In Node.js, some of these principles come directly from its creator, Ryan Dahl; from all the people who contributed to the core; from charismatic figures in the community; and some of the principles are inherited from the JavaScript culture or are influenced by the Unix philosophy.

None of these rules are imposed and they should always be applied with common sense; however, they can prove to be tremendously useful when we are looking for a source of inspiration while designing our programs.

You can find an extensive list of software development philosophies on Wikipedia at `http://en.wikipedia.org/wiki/List_of_software_development_philosophies`.

Small core

The Node.js core itself has its foundations built on a few principles; one of these is having the smallest set of functionalities, leaving the rest to the so-called **userland** (or **userspace**), the ecosystem of modules living outside the core. This principle has an enormous impact on the Node.js culture, as it gives freedom to the community to experiment and iterate quickly on a broader set of solutions within the scope of the userland modules, instead of being imposed with one slowly evolving solution that is built into the more tightly controlled and stable core. Keeping the core set of functionalities to the bare minimum, then, not only becomes convenient in terms of maintainability, but also in terms of the positive cultural impact that it brings on the evolution of the entire ecosystem.

Small modules

Node.js uses the concept of a *module* as a fundamental means to structure the code of a program. It is the building block for creating applications and reusable libraries called *packages* (a package is also frequently referred to as a module since, usually, it has one single module as an entry point). In Node.js, one of the most evangelized principles is to design small modules, not only in terms of code size, but most importantly in terms of scope.

This principle has its roots in the Unix philosophy, particularly in two of its precepts, which are as follows:

- "Small is beautiful."
- "Make each program do one thing well."

Node.js brought these concepts to a whole new level. Along with the help of npm, the official package manager, Node.js helps solve the *dependency hell* problem by making sure that each installed package will have its own separate set of dependencies, thus enabling a program to depend on a lot of packages without incurring conflicts. The Node way, in fact, involves extreme levels of reusability, whereby applications are composed of a high number of small, well-focused dependencies. While this can be considered unpractical or even totally unfeasible in other platforms, in Node.js this practice is encouraged. As a consequence, it is not rare to find npm packages containing less than 100 lines of code or exposing only one single function.

Besides the clear advantage in terms of reusability, a small module is also considered to be the following:

- Easier to understand and use
- Simpler to test and maintain
- Perfect to share with the browser

Having smaller and more focused modules empowers everyone to share or reuse even the smallest piece of code; it's the **Don't Repeat Yourself** (**DRY**) principle applied to a whole new level.

Small surface area

In addition to being small in size and scope, Node.js modules usually also have the characteristic of exposing a minimal set of functionalities. The main advantage here is increased usability of the API, which means that the API becomes clearer to use and is less exposed to erroneous usage. Most of the time, in fact, the user of a component is only interested in a very limited and focused set of features, without the need to extend its functionality or tap into more advanced aspects.

In Node.js, a very common pattern for defining modules is to expose only one piece of functionality, such as a function or a constructor, while letting more advanced aspects or secondary features become properties of the exported function or constructor. This helps the user to identify what is important and what is secondary. It is not rare to find modules that expose only one function and nothing else, for the simple fact that it provides a single, unmistakably clear entry point.

Another characteristic of many Node.js modules is the fact that they are created to be used rather than extended. Locking down the internals of a module by forbidding any possibility of an extension might sound inflexible, but it actually has the advantage of reducing the use cases, simplifying its implementation, facilitating its maintenance, and increasing its usability.

Simplicity and pragmatism

Have you ever heard of the **Keep It Simple, Stupid** (**KISS**) principle or the famous quote:

> *"Simplicity is the ultimate sophistication."*
>
> *– Leonardo da Vinci*

Richard P. Gabriel, a prominentcomputer scientist, coined the term "worse is better" to describe the model, whereby less and simpler functionality is a good design choice for software. In his essay, *The Rise of "Worse is Better"*, he says:

> *"The design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design."*

Designing simple, as opposed to perfect, fully-featured software, is a good practice for several reasons: it takes less effort to implement, allows faster shipping with fewer resources, is easier to adapt, and is easier to maintain and understand. These factors foster community contributions and allow the software itself to grow and improve.

In Node.js, this principle is also enabled by JavaScript, which is a very pragmatic language. It's not rare, in fact, to see simple functions, closures, and object literals replacing complex class hierarchies. Pure object-oriented designs often try to replicate the real world using the mathematical terms of a computer system without considering the imperfection and the complexity of the real world itself. The truth is that; our software is always an approximation of reality, and we would probably have more success in trying to get something working sooner and with reasonable complexity, instead of trying to create near-perfect software with huge effort and tons of code to maintain.

Throughout this book, we will see this principle in action many times. For example, a considerable number of traditional design patterns, such as singleton or decorator, can have a trivial, even if sometimes not foolproof, implementation and we will see how an uncomplicated, practical approach (most of the time) is preferred to a pure, flawless design.

Introduction to Node.js 6 and ES2015

At the time of writing, the latest major releases of Node.js (versions 4, 5, and 6) come with the great addition of increased language support for the new features introduced in the ECMAScript 2015 specification (in short, ES2015, and formerly known also as ES6), which aims to make the JavaScript language even more flexible and enjoyable.

Throughout this book, we will widely adopt some of these new features in the code examples. These concepts are still fresh within the Node.js community so it's worth having a quick look at the most important ES2015-specific features currently supported in Node.js. Our version of reference is Node.js version 6.

Depending on your Node.js version, some of these features will work correctly only when **strict mode** is enabled. Strict mode can be easily enabled by adding a `"use strict"` statement at the very beginning of your script. Notice that the `"use strict"` statement is a plain string and that you can either use single or double quotes to declare it. For the sake of brevity, we will not write this line in our code examples, but you should remember to add it to be able to run them correctly.

The following list is not meant to be exhaustive but just an introduction to some of the ES2015 features supported in Node.js, so that you can easily understand all the code examples in the rest of the book.

The let and const keywords

Historically, JavaScript only offered function scope and global scope to control the lifetime and the visibility of a variable. For instance, if you declare a variable inside the body of an `if` statement, the variable will be accessible even outside the statement, whether or not the body of the statement has been executed. Let's see it more clearly with an example:

```
if (false) {
   var x = "hello";
}
console.log(x);
```

This code will not fail as we might expect and it will just print `undefined` in the console. This behavior has been the cause of many bugs and a lot of frustration, and that is the reason why ES2015 introduces the `let` keyword to declare variables that respect the block scope. Let's replace`var` with `let` in our previous example:

```
if (false) {
   let x = "hello";
}
console.log(x);
```

This code will raise a `ReferenceError: x is not defined` because we are trying to print a variable that has been defined inside another block.

To give a more meaningful example we can use the `let` keyword to define a temporary variable to be used as an index for a loop:

```
for (let i=0; i < 10; i++) {
  // do something here
}
console.log(i);
```

As in the previous example, this code will raise a `ReferenceError: i is not defined` error.

This protective behavior introduced with `let` allows us to write safer code, because if we accidentally access variables that belong to another scope, we will get an error that will allow us to easily spot the bug and avoid potentially dangerous side effects.

ES2015 introduces also the `const` keyword. This keyword allows us to declare constant variables. Let's see a quick example:

```
const x = 'This will never change';
x = '...';
```

This code will raise a `TypeError: Assignment to constant variable` error because we are trying to change the value of a constant.

Anyway, it's important to underline that `const` does not behave in the same way as constant values in many other languages where this keyword allows us to define read-only variables. In fact, in ES2015, `const` does not indicate that the assigned value will be constant, but that the binding with the value is constant. To clarify this concept, we can see that with `const` in ES2015 it is still possible to do something like this:

```
const x = {};
x.name = 'John';
```

When we change a property inside the object we are actually altering the value (the object), but the binding between the variable and the object will not change, so this code will not raise an error. Conversely, if we reassign the full variable, this will change the binding between the variable and its value and raise an error:

```
x = null; // This will fail
```

Constants are extremely useful when you want to protect a scalar value from being accidentally changed in your code or, more generically, when you want to protect an assigned variable to be accidentally reassigned to another value somewhere else in your code.

It is becoming best practice to use `const` when requiring a module in a script, so that the variable holding the module cannot be accidentally reassigned:

```
const path = require('path');
// .. do stuff with the path module
let path = './some/path'; // this will fail
```

If you want to create an immutable object, `const` is not enough, so you should use ES5's method `Object.freeze()` (`https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze`) or the `deep-freeze` module (`https://www.npmjs.com/package/deep-freeze`).

The arrow function

One of the most appreciated features introduced by ES2015 is the support for arrow functions. The arrow function is a more concise syntax for defining functions, especially useful when defining a callback. To better understand the advantages of this syntax, let's first see an example of classic filtering on an array:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(function(x) {
  return x%2 === 0;
});
```

The preceding code can be rewritten as follows using the arrow function syntax:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(x => x%2 === 0);
```

The `filter` function can be defined inline, and the keyword `function` is removed, leaving only the list of parameters, which is followed by `=>` (the arrow), which in turn is followed by the body of the function. When the list of arguments contains more than one argument, you must surround them with parentheses and separate the argument with commas. Also, when there is no argument you must provide a set of empty parentheses before the arrow: `() => {...}`. When the body of the function is just one line, there's no need to write the `return` keyword as it is applied implicitly. If we need to add more lines of code to the body of the function, we can wrap them in curly brackets, but beware that in this case `return` is not automatically implied, so it needs to be stated explicitly, as in the following example:

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(x => {
  if (x%2 === 0) {
    console.log(x + ' is even!');
    return true;
  }
});
```

But there is another important feature to know about arrow functions: arrow functions are bound to their lexical scope. This means that inside an arrow function the value of `this` is the same as in the parent block. Let's clarify this concept with an example:

```
function DelayedGreeter(name) {
  this.name = name;
}

DelayedGreeter.prototype.greet = function() {
  setTimeout( function cb() {
    console.log('Hello ' + this.name);
```

```
  }, 500);
};

const greeter = new DelayedGreeter('World');
greeter.greet(); // will print "Hello undefined"
```

In this code, we are defining a simple `greeter` prototype that accepts a name as an argument. Then we are adding the `greet` method to the prototype. This function is supposed to print `Hello` and the name defined in the current instance `500` milliseconds after it has been called. But this function is broken, because inside the timeout callback function `(cb)`, the scope of the function is different from the scope of `greet` method and the value of `this` is `undefined`.

Before Node.js introduced support for arrow functions, to fix this we needed to change the `greet` function using `bind`, as follows:

```
DelayedGreeter.prototype.greet = function() {
  setTimeout( (function cb() {
    console.log('Hello' + this.name);
  }).bind(this), 500);
};
```

But since we have now arrow functions and since they are bound to their lexical scope, we can just use an arrow function as a callback to solve the issue:

```
DelayedGreeter.prototype.greet = function() {
  setTimeout( () => console.log('Hello' + this.name), 500);
};
```

This is a very handy feature; most of the time it makes our code more concise and straightforward.

Class syntax

ES2015 introduces a new syntax to leverage prototypical inheritance in a way that should sound more familiar to all the developers that come from classic object-oriented languages such as Java or C#. It's important to underline that this new syntax does not change the way objects are managed internally by the JavaScript runtime; they still inherit properties and functions through prototypes and not through classes. While this new alternative syntax can be very handy and readable, as a developer, it is important to understand that it is just syntactic sugar.

Let's see how it works with a trivial example. First of all, let's describe a `Person` function using the classic prototype-based approach:

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

Person.prototype.getFullName = function() {
  return this.name + '' + this.surname;
};

Person.older = function(person1, person2) {
  return (person1.age >= person2.age) ? person1 : person2;
};
```

As you can see, a person has `name`, `surname`, and `age`. We are providing our prototype with a helper function that allows us to easily get the full name of a `person` object and a generic helper function accessible directly from the `Person` prototype that returns the older person between two `Person` instances given as input.

Let's see now how we can implement the same example using the new handy ES2015class syntax:

```
class Person {
  constructor (name, surname, age) {
    this.name = name;
    this.surname = surname;
    this.age = age;
  }

  getFullName () {
    return this.name + ' ' + this.surname;
  }

  static older (person1, person2) {
    return (person1.age >= person2.age) ? person1 : person2;
  }
}
```

This syntax is more readable and straightforward to understand. We are explicitly stating what the `constructor` is for the class and declaring the function `older` as a `static` method.

The two implementations are completely interchangeable, but the real killer feature of the new syntax is the possibility of extending the `Person` prototype using the `extend` and `super` keywords. Let's assume we want to create a `PersonWithMiddlename` class:

```
class PersonWithMiddlename extends Person {
  constructor (name, middlename, surname, age) {
    super(name, surname, age);
    this.middlename = middlename;
  }

  getFullName () {
    return this.name + '' + this.middlename + '' + this.surname;
  }
}
```

What is worth noticing in this third example is that the syntax really resembles what is common in other object-oriented languages. We are declaring the class from which we want to extend, we define a new constructor that can call the parent one using the keyword `super`, and we override the `getFullName` method to add support for our middle name.

Enhanced object literals

Along with the new class syntax, ES2015 introduced an enhanced object literals syntax. This syntax offers a shorthand to assign variables and functions as members of the object, allows us to define computed member names at creation time, and also handy setter and getter methods.

Let's make all of this clear with some examples:

```
const x = 22;
const y = 17;
const obj = { x, y };
```

`obj` will be an object containing the keys `x` and `y` with the values `22` and `17`, respectively.

We can do the same thing with functions:

```
module.exports = {
  square (x) {
    return x * x;
  },
  cube (x) {
    return x * x * x;
  }
};
```

In this case, we are writing a module that exports the functions `square` and `cube` mapped to properties with the same name. Notice that we don't need to specify the keyword `function`.

Let's see in another example how we can use computed property names:

```
const namespace = '-webkit-';
const style = {
  [namespace + 'box-sizing'] : 'border-box',
  [namespace + 'box-shadow'] : '10px10px5px #888888'
};
```

In this case, the resulting object will contain the properties `-webkit-box-sizing` and `-webkit-box-shadow`.

Let's see now how we can use the new setter and getter syntax by jumping directly to an example:

```
const person = {
  name : 'George',
  surname : 'Boole',

  get fullname () {
    return this.name + '' + this.surname;
  },

  set fullname (fullname) {
    let parts = fullname.split('');
    this.name = parts[0];
    this.surname = parts[1];
  }
};

console.log(person.fullname); // "George Boole"
console.log(person.fullname = 'Alan Turing'); // "Alan Turing"
console.log(person.name); // "Alan"
```

In this example we are defining three properties, two normal ones, `name` and `surname`, and a computed `fullname` property through the `set` and `get` syntax. As you can see from the result of the `console.log` calls, we can access the computed property as if it was a regular property inside the object for both reading and writing the value. It's worth noticing that the second call to `console.log` prints `Alan Turing`. This happens because by default every `set` function returns the value that is returned by the `get` function for the same property, in this case `get fullname`.

Map and Set collections

As JavaScript developers, we are used to creating hash maps using plain objects. ES2015 introduces a new prototype called `Map` that is specifically designed to leverage hash map collections in a more secure, flexible, and intuitive way. Let's see a quick example:

```
const profiles = new Map();
profiles.set('twitter', '@adalovelace');
profiles.set('facebook', 'adalovelace');
profiles.set('googleplus', 'ada');

profiles.size; // 3
profiles.has('twitter'); // true
profiles.get('twitter'); // "@adalovelace"
profiles.has('youtube'); // false
profiles.delete('facebook');
profiles.has('facebook'); // false
profiles.get('facebook'); // undefined
for (const entry of profiles) {
  console.log(entry);
}
```

As you can see, the `Map` prototype offers several handy methods, such as `set`, `get`, `has`, and `delete`, and the `size` attribute (notice how the latter differs from arrays where we use the attribute `length`). We can also iterate through all the entries using the `for...of` syntax. Every entry in the loop will be an array containing the key as first element and the value as second element. This interface is very intuitive and self-explanatory.

But what makes maps really interesting is the possibility of using functions and objects as keys of the map, and this is something that is not entirely possible using plain objects, because with objects all the keys are automatically cast to strings. This opens new opportunities; for example, we can build a micro testing framework leveraging this feature:

```
const tests = new Map();
tests.set(() => 2+2, 4);
tests.set(() => 2*2, 4);
tests.set(() => 2/2, 1);

for (const entry of tests) {
  console.log((entry[0]() === entry[1]) ? 'PASS' : 'FAIL');
}
```

As you can see in this last example, we are storing functions as keys and expected results as values. Then we can iterate through our hash map and execute all the functions. It's also worth noticing that when we iterate through the map, all the entries respect the order in which they have been inserted; this is also something that was not always guaranteed with plain objects.

Along with `Map`, ES2015 also introduces the `Set` prototype. This prototype allows us to easily construct sets, which means lists with unique values:

```
const s = new Set([0, 1, 2, 3]);
s.add(3); // will not be added
s.size; // 4
s.delete(0);
s.has(0); // false

for (const entry of s) {
  console.log(entry);
}
```

As you can see, in this example the interface is quite similar to the one we have just seen for `Map`. We have the methods `add` (instead of `set`), `has`, and `delete` and the property `size`. We can also iterate through the set and in this case every entry is a value, in our example it will be one of the numbers in the set. Finally, sets can also contain objects and functions as values.

WeakMap and WeakSet collections

ES2015 also defines a "weak" version of the `Map` and the `Set` prototypes called `WeakMap` and `WeakSet`.

`WeakMap` is quite similar to `Map` in terms of interface; however, there are two main differences you should be aware of: there is no way to iterate all over the entries, and it only allows having objects as keys. While this might seem like a limitation, there is a good reason behind it. In fact, the distinctive feature of `WeakMap` is that it allows objects used as keys to be garbage collected when the only reference left is inside `WeakMap`. This is extremely useful when we are storing some metadata associated with an object that might get deleted during the regular lifetime of the application. Let's see an example:

```
let obj = {};
const map = new WeakMap();
map.set(obj, {key: "some_value"});
console.log(map.get(obj)); // {key: "some_value"}
obj = undefined; // now obj and the associated data in the map
```

```
// will be cleaned up in the next gc cycle
```

In this code, we are creating a plain object called `obj`. Then we store some metadata for this object in a new `WeakMap` called `map`. We can access this metadata with the `map.get` method. Later, when we cleanup the object by assigning its variable to `undefined`, the object will be correctly garbage collected and its metadata removed from the map.

Similar to `WeakMap`, `WeakSet` is the weak version of `Set`: it exposes the same interface of `Set` but it only allows storing objects and cannot be iterated. Again, the difference with `Set` is that `WeakSet` allows objects to be garbage collected when their only reference left is in the weak set:

```
let obj1= {key: "val1"};
let obj2= {key: "val2"};
const set= new WeakSet([obj1, obj2]);
console.log(set.has(obj1)); // true
obj1= undefined; // now obj1 will be removed from the set
console.log(set.has(obj1)); // false
```

It's important to understand that `WeakMap` and `WeakSet` are not better or worse than `Map` and `Set`, they are simply more suitable for different use cases.

Template literals

ES2015 offers a new alternative and more powerful syntax to define strings: the `template` literals. This syntax uses back ticks (`` ` ``) as delimiters and offers several benefits compared to regular quoted (`'`) or double-quoted (`"`) delimited strings. The main benefits are that template literal syntax can interpolate variables or expressions using `${expression}` inside the string (this is the reason why this syntax is called "template") and that a single string can finally be easily written in multiple lines. Let's see a quick example:

```
const name = "Leonardo";
const interests = ["arts", "architecture", "science", "music",
                   "mathematics"];
const birth = { year : 1452, place : 'Florence' };
const text = `${name} was an Italian polymath
 interested in many topics such as
 ${interests.join(', ')}.He was born
 in ${birth.year} in ${birth.place}.`;
console.log(text);
```

This code will print the following:

```
Leonardo was an Italian polymath interested in many topics
such as arts, architecture, science, music, mathematics.
He was born in 1452 in Florence.
```

Downloading the example code
Detailed steps to download the code bundle are mentioned in the *Preface* of this book. Have a look.
The code bundle for the book is also hosted on GitHub at:
`http://bit.ly/node_book_code`.
We also have other code bundles from our rich catalog of books and videos available at:
`https://github.com/PacktPublishing/`.

Other ES2015 features

Another extremely interesting feature added in ES2015 and available since Node.js version 4 is **Promise**. We will discuss Promise in detail in `Chapter 4`, *Asynchronous Control Flow Patterns with ES2015 and Beyond*.

Other interesting ES2015 features introduced in Node.js version 6 are as follows:

- Default function parameters
- Rest parameters
- Spread operator
- Destructuring
- `new.target` (we will talk about this in `Chapter 2`, *Node.js Essential Patterns*)
- Proxy (we will talk about this in `Chapter 6`, *Design Patterns*)
- Reflect
- Symbols

A more extended and up-to-date list of all the supported ES2015 features is available in the official Node.js documentation:
`https://nodejs.org/en/docs/es6/`.

The reactor pattern

In this section, we will analyze the reactor pattern, which is the heart of the asynchronous nature of Node.js. We will go through the main concepts behind the pattern, such as the single-threaded architecture and the non-blocking I/O, and we will see how this creates the foundation for the entire Node.js platform.

I/O is slow

I/O is definitely the slowest among the fundamental operations of a computer. Accessing the RAM is in the order of nanoseconds (10E-9 seconds), while accessing data on the disk or the network is in the order of milliseconds (10E-3 seconds). For the bandwidth, it is the same story; RAM has a transfer rate consistently in the order of GB/s, while disk and network varies from MB/s to, optimistically, GB/s. I/O is usually not expensive in terms of CPU, but it adds a delay between the moment the request is sent and the moment the operation completes. On top of that, we also have to consider the human factor; often, the input of an application comes from a real person, for example, the click of a button or a message sent in a real-time chat application, so the speed and frequency of I/O doesn't only depend on technical aspects and it can be many orders of magnitude slower than the disk or network.

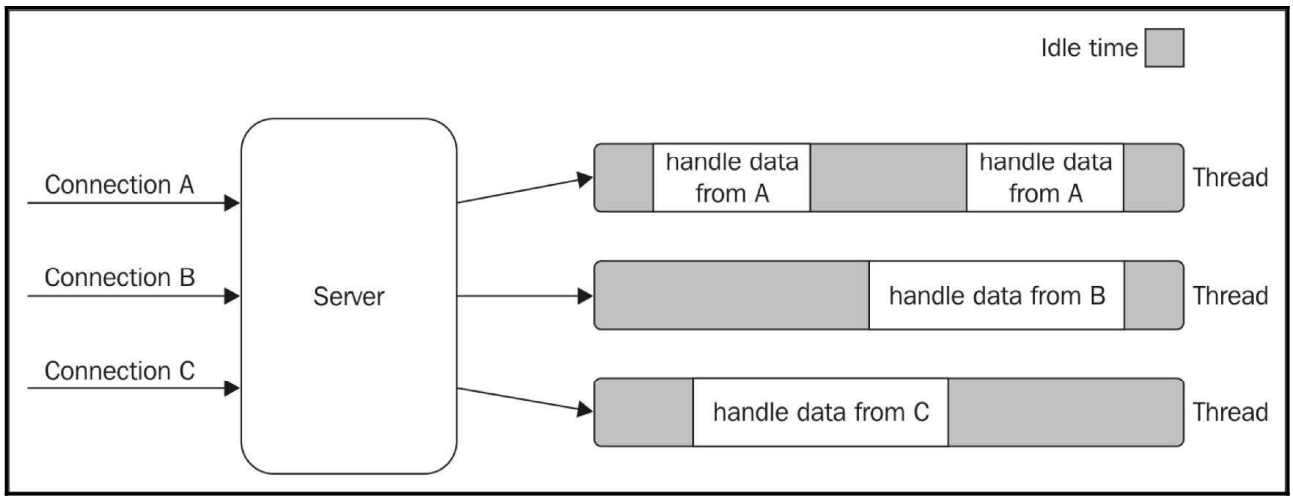
Blocking I/O

In traditional blocking I/O programming, the function call corresponding to an I/O request will block the execution of the thread until the operation completes. This can go from a few milliseconds, in the case of disk access, to minutes or even more, in case the data is generated from user actions, such as pressing a key. The following pseudocode shows a typical blocking thread performed against a socket:

```
//blocks the thread until the data is available
data = socket.read();
//data is available
print(data);
```

It is trivial to notice that a web server that is implemented using blocking I/O will not be able to handle multiple connections in the same thread; each I/O operation on a socket will block the processing of any other connection. For this reason, the traditional approach to handling concurrency in web servers is to kick off a thread or a process (or to reuse one taken from a pool) for each concurrent connection that needs to be handled. This way, when a thread gets blocked for an I/O operation it will not impact the availability of the other requests, because they are handled in separate threads.

The following image illustrates this scenario:



The preceding image lays emphasis on the amount of time each thread is idle, waiting for new data to be received from the associated connection. Now, if we also consider that any type of I/O can possibly block a request, for example, while interacting with databases or with the filesystem, we soon realize how many times a thread has to block in order to wait for the result of an I/O operation. Unfortunately, a thread is not cheap in terms of system resources; it consumes memory and causes context switches, so having a long-running thread for each connection and not using it for most of the time is not the best compromise in terms of efficiency.

Non-blocking I/O

In addition to blocking I/O, most modern operating systems support another mechanism to access resources called non-blocking I/O. In this operating mode, the system call always returns immediately without waiting for the data to be read or written. If no results are available at the moment of the call, the function will simply return a predefined constant, indicating that there is no data available to return at that moment.

For example, in Unix operating systems, the `fcntl()` function is used to manipulate an existing file descriptor to change its operating mode to non-blocking (with the `O_NONBLOCK` flag). Once the resource is in non-blocking mode, any read operation will fail with the return code `EAGAIN`, in case the resource doesn't have any data ready to be read.

The most basic pattern for accessing this kind of non-blocking I/O is to actively poll the resource within a loop until some actual data is returned; this is called **busy-waiting**. The following pseudocode shows you how it's possible to read from multiple resources using non-blocking I/O and a polling loop:

```
resources = [socketA, socketB, pipeA];
while(!resources.isEmpty()) {
  for(i = 0; i < resources.length; i++) {
    resource = resources[i];
    //try to read
    let data = resource.read();
    if(data === NO_DATA_AVAILABLE)
      //there is no data to read at the moment
      continue;
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the list
      resources.remove(i);
    else
      //some data was received, process it
      consumeData(data);
  }
}
```

You can see that, with this simple technique, it is already possible to handle different resources in the same thread, but it's still not efficient. In fact, in the preceding example, the loop will only consume precious CPU for iterating over resources that are unavailable most of the time. Polling algorithms usually result in a huge amount of wasted CPU time.

Event demultiplexing

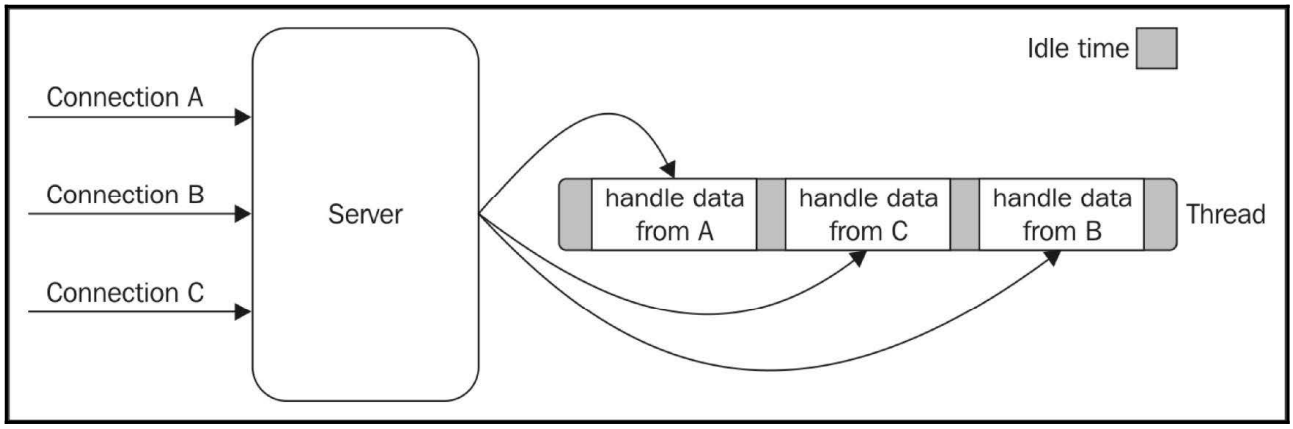
Busy-waiting is definitely not an ideal technique for processing non-blocking resources, but luckily, most modern operating systems provide a native mechanism to handle concurrent, non-blocking resources in an efficient way; this mechanism is called **synchronous event demultiplexer** or **event notification interface**. This component collects and queues I/O events that come from a set of watched resources, and block until new events are available to process. The following is the pseudocode of an algorithm that uses a generic synchronous event demultiplexer to read from two different resources:

```
socketA, pipeB;
watchedList.add(socketA, FOR_READ);                    //[1]
watchedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(watchedList)) {     //[2]
  //event loop
  foreach(event in events) {                           //[3]
    //This read will never block and will always return data
    data = event.resource.read();
    if(data === RESOURCE_CLOSED)
      //the resource was closed, remove it from the watched list
      demultiplexer.unwatch(event.resource);
    else
      //some actual data was received, process it
      consumeData(data);
  }
}
```

These are the important steps of the preceding pseudocode:

1. The resources are added to a data structure, associating each one of them with a specific operation, in our example, `read`.
2. The event notifier is set up with the group of resources to be watched. This call is synchronous and blocks until any of the watched resources are ready for `read`. When this occurs, the event demultiplexer returns from the call and a new set of events is available to be processed.
3. Each event returned by the event demultiplexer is processed. At this point, the resource associated with each event is guaranteed to be ready to read and to not block during the operation. When all the events are processed, the flow will block again on the event demultiplexer until new events are again available to be processed. This is called the **event loop**.

It's interesting to see that with this pattern, we can now handle several I/O operations inside a single thread, without using a busy-waiting technique. The following image shows us how a web server would be able to handle multiple connections using a synchronous event demultiplexer and a single thread:

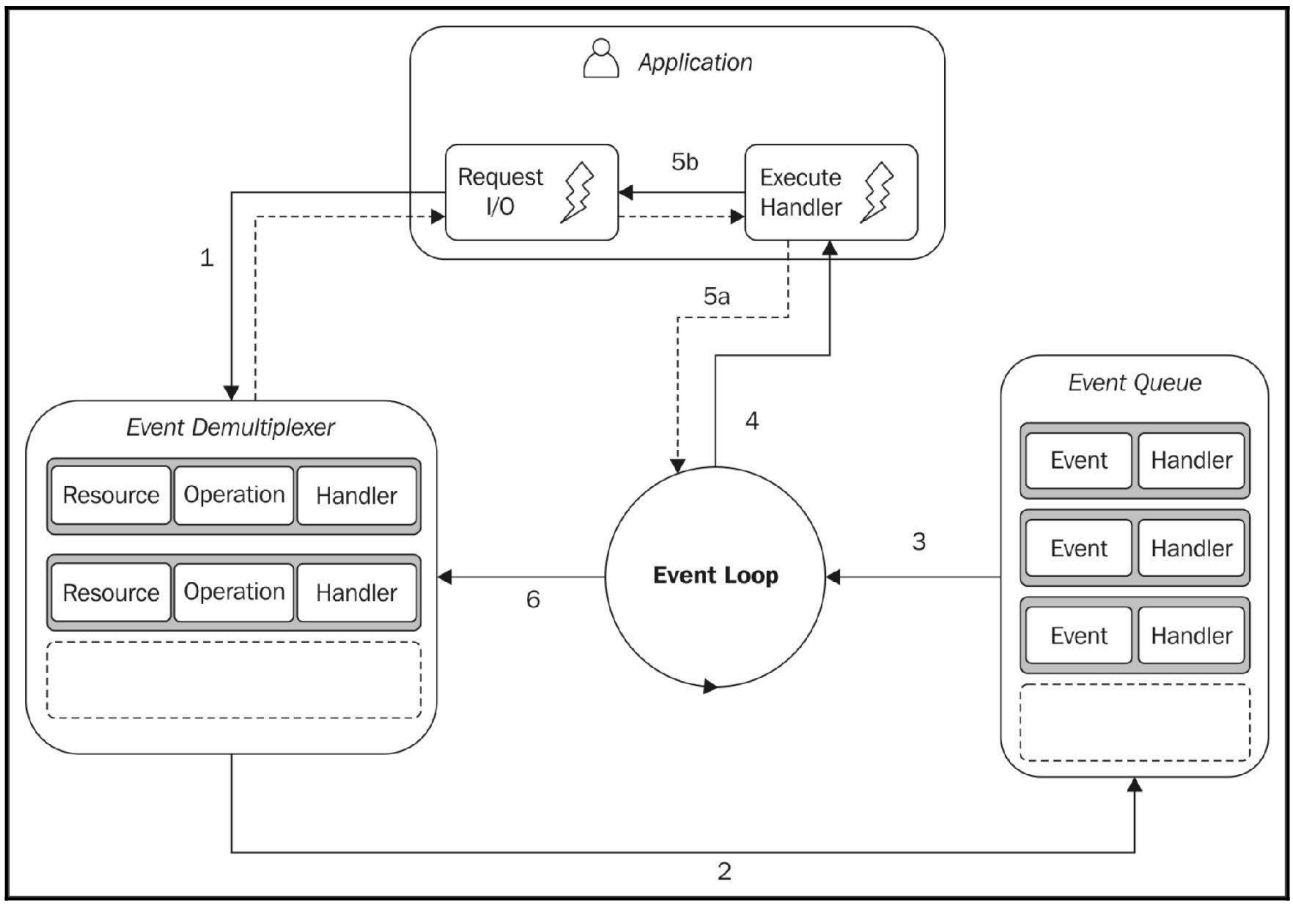


The previous image helps us understand how concurrency works in a single-threaded application using a synchronous event demultiplexer and non-blocking I/O. We can see that using only one thread does not impair our ability to run multiple I/O bound tasks concurrently. The tasks are spread over time, instead of being spread across multiple threads. This has the clear advantage of minimizing the total idle time of the thread, as clearly shown in the image. This is not the only reason for choosing this model. To have only a single thread, in fact, also has a beneficial impact on the way programmers approach concurrency in general. Throughout the book, we will see how the absence of in-process race conditions and multiple threads to synchronize allows us to use much simpler concurrency strategies.

In the next chapter, we will have the opportunity to talk more about the concurrency model of Node.js.

Introducing the reactor pattern

We can now introduce the reactor pattern, which is a specialization of the algorithms presented in the previous section. The main idea behind it is to have a handler (which in Node.js is represented by a `callback` function) associated with each I/O operation, which will be invoked as soon as an event is produced and processed by the event loop. The structure of the reactor pattern is shown in the following image:



This is what happens in an application using the reactor pattern:

1. The application generates a new I/O operation by submitting a request to the **Event Demultiplexer**. The application also specifies a handler, which will be invoked when the operation completes. Submitting a new request to the **Event Demultiplexer** is a non-blocking call and it immediately returns control to the application.
2. When a set of I/O operations completes, the **Event Demultiplexer** pushes the new events into the **Event Queue**.
3. At this point, the **Event Loop** iterates over the items of the **Event Queue**.
4. For each event, the associated handler is invoked.
5. The handler, which is part of the application code, will give back control to the **Event Loop** when its execution completes (**5a**). However, new asynchronous operations might be requested during the execution of the handler (**5b**), causing new operations to be inserted in the **Event Demultiplexer** (**1**), before control is given back to the **Event Loop**.
6. When all the items in the **Event Queue** are processed, the loop will block again on the **Event Demultiplexer** which will then trigger another cycle when a new event is available.

The asynchronous behavior is now clear: the application expresses the interest to access a resource at one point in time (without blocking) and provides a handler, which will then be invoked at another point in time when the operation completes.

A Node.js application will exit automatically when there are no more pending operations in the Event Demultiplexer, and no more events to be processed inside the **Event Queue**.

We can now define the pattern at the heart of Node.js:

Pattern (reactor) handles I/O by blocking until new events are available from a set of observed resources, and then reacts by dispatching each event to an associated handler.

The non-blocking I/O engine of Node.js-libuv

Each operating system has its own interface for the **Event Demultiplexer:** epoll on Linux, kqueue on Mac OS X, and the **I/O Completion Port** (**IOCP**) API on Windows. Besides that, each I/O operation can behave quite differently depending on the type of the resource, even within the same OS. For example, in Unix, regular filesystem files do not support non-blocking operations, so, in order to simulate non-blocking behavior, it is necessary to use a separate thread outside the Event Loop. All these inconsistencies across and within the different operating systems required a higher-level abstraction to be built for the Event Demultiplexer. This is exactly why the Node.js core team created a C library called **libuv**, with the objective to make Node.js compatible with all the major platforms and normalize the non-blocking behavior of the different types of resource; libuv today represents the low-level I/O engine of Node.js.

Besides abstracting the underlying system calls, libuv also implements the reactor pattern, thus providing an API for creating event loops, managing the event queue, running asynchronous I/O operations, and queuing other types of task.

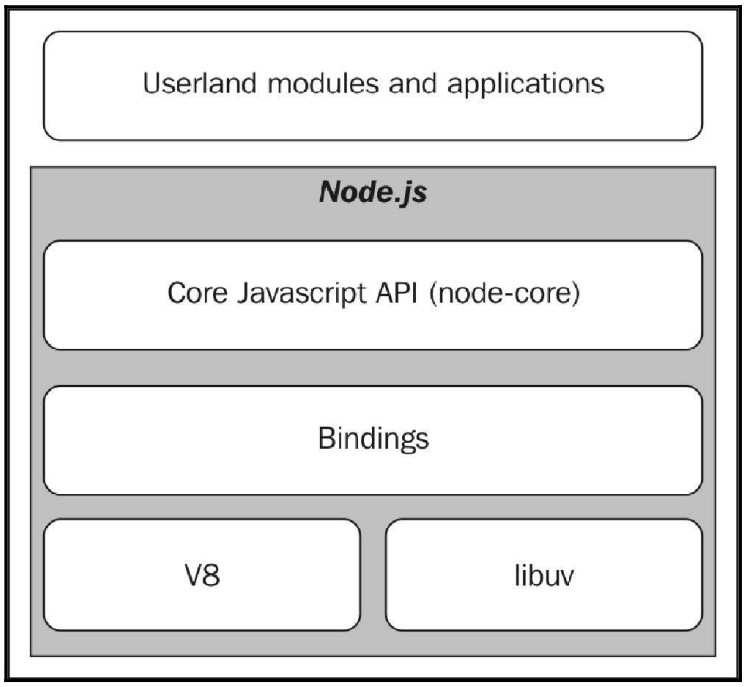
A great resource to learn more about libuv is the free online book created by Nikhil Marathe, which is available at:
`http://nikhilm.github.io/uvbook/`

The recipe for Node.js

The reactor pattern and libuv are the basic building blocks of Node.js, but we need the following three other components to build the full platform:

- A set of bindings responsible for wrapping and exposing libuv and other low-level functionality to JavaScript.
- **V8**, the JavaScript engine originally developed by Google for the Chrome browser. This is one of the reasons why Node.js is so fast and efficient. V8 is acclaimed for its revolutionary design, its speed, and for its efficient memory management.
- A core JavaScript library (called **node-core**) that implements the high-level Node.js API.

Finally, this is the recipe of Node.js, and the following image represents its final architecture:



Summary

In this chapter, we have seen how the Node.js platform is based on a few important principles that provide the foundation to build efficient and reusable code. The philosophy and the design choices behind the platform have, in fact, a strong influence on the structure and behavior of every application and module we create. Often, for a developer moving from another technology, these principles might seem unfamiliar and the usual instinctive reaction is to fight the change by trying to find more familiar patterns inside a world which, in reality, requires a real shift in the mindset.

On one hand, the asynchronous nature of the reactor pattern requires a different programming style made of callbacks and things that happen at a later time, without worrying too much about threads and race conditions. On the other hand, the module pattern and its principles of simplicity and minimalism create interesting new scenarios in terms of reusability, maintenance, and usability.

Finally, besides the obvious technical advantages of being fast, efficient, and based on JavaScript, Node.js is attracting so much interest because of the principles we have just discovered. For many, grasping the essence of this world feels like returning to the origins, to a more humane way of programming in both size and complexity, and that's why developers end up falling in love with Node.js. The introduction of ES2015 makes things even more interesting and opens new scenarios in which we can embrace all these advantages with an even more expressive syntax.

In the next chapter, we will get deep into the two basic asynchronous patterns used in Node.js: the callback pattern and the event emitter. We will also understand the difference between synchronous and asynchronous code and how to avoid writing unpredictable functions.