In this section, you will gain an understanding of how Node.js works internally and be introduced to 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 you will see how this creates the foundation for the entire Node.js platform.

I/O is slow

I/O (short for input/output) 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). The same applies to the bandwidth. RAM has a transfer rate consistently in the order of GB/s, while the disk or 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 to the device and the moment the operation completes. On top of that, we have to consider the human factor. In fact, in many circumstances, the input of an application comes from a real person—a mouse click, for example—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 range from a few milliseconds, in the case of disk access, to minutes or even more, in the case of data being 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
```

> *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, it allows shipping faster with fewer resources, it's easier to adapt, and, finally, it's easier to maintain and understand. The positive effects of these factors encourage community contributions and allow the software itself to grow and improve.

In Node.js, the adoption of this principle is also facilitated by JavaScript, which is a very pragmatic language. In fact, it's common to see simple classes, functions, and closures 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 complexity of the real world itself. Instead, the truth is that our software is always an approximation of reality, and we will probably have more success by 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, you 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 bulletproof, implementation, and you will see how an uncomplicated, practical approach is (most of the time) preferred to a pure, flawless design.

Next, we will take a look inside the Node.js core to reveal its internal patterns and event-driven architecture.

How Node.js works

Small surface area

In addition to being small in size and scope, a desirable characteristic of Node.js modules is exposing a minimal set of functionalities to the outside world. This has the effect of producing an API that is clearer to use and less susceptible to erroneous usage. In fact, most of the time the user of a component is only interested in a very limited and focused set of features, without needing 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 functionality, such as a function or a class, 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 use cases, simplifying implementation, facilitating maintenance, and increasing usability. In practice, this means preferring to expose functions instead of classes, and being careful not to expose any internals to the outside world.

Simplicity and pragmatism

Have you ever heard of the **Keep It Simple, Stupid** (**KISS**) principle? Richard P. Gabriel, a prominent computer 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.*

Small modules

Node.js uses the concept of a **module** as the fundamental means for structuring the code of a program. It is the building block for creating applications and reusable libraries. In Node.js, one of the most evangelized principles is designing small modules (and packages), not only in terms of raw code size, but, most importantly, in terms of scope.

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

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

Node.js has brought these concepts to a whole new level. Along with the help of its module managers—with **npm** and **yarn** being the most popular—Node.js helps to solve the *dependency hell* problem by making sure that two (or more) packages depending on different versions of the same package will use their own installations of such a package, thus avoiding conflicts. This aspect allows packages to depend on a high number of small, well-focused dependencies without the risk of creating conflicts. While this can be considered unpractical or even totally unfeasible in other platforms, in Node.js, this practice is the norm. This enables extreme levels of reusability; they are so extreme, in fact, that sometimes we can find packages comprising of a single module containing just a couple of lines of code—for example, a regular expression for matching emails such as `nodejsdp.link/email-regex`.

Besides the clear advantage in terms of reusability, a small module is also:

- Easier to understand and use
- Simpler to test and maintain
- Small in size and perfect for use in 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 at a whole new level.

Every programming platform has its own philosophy, a set of principles and guidelines that are generally accepted by the community, or an ideology for doing things that influence both the evolution of the 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 ideologies borrowed from other platforms. In Node.js, some of these principles come directly from its creator—Ryan Dahl—while others come from the people who contribute to the core or from charismatic figures in the community, and, finally, some are inherited from the larger JavaScript movement.

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 software.

You can find an extensive list of software development philosophies on Wikipedia at nodejsdp.link/dev-philosophies.

Small core

The Node.js core—understood as the Node.js runtime and built-in modules—has its foundations built on a few principles. One of these is having the smallest possible set of functionalities, while leaving the rest to the so-called **userland** (or **userspace**), which is 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 having 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, is not only convenient in terms of maintainability, but also in terms of the positive cultural impact that it brings to the evolution of the entire ecosystem.

1

The Node.js Platform

Some principles and design patterns literally define the developer experience with the Node.js platform and its ecosystem. The most peculiar one is probably its asynchronous nature, which makes heavy use of asynchronous constructs such as callbacks and promises. In this introductory chapter, we will explore where Node.js gets its asynchronous behavior from. This is not just good-to-know theoretical information: knowing how Node.js works at its core will give you a strong foundation for understanding the reasoning behind more complex topics and patterns that we will cover later in the book.

Another important aspect that characterizes Node.js is its philosophy. Approaching Node.js is, in fact, far more than simply learning a new technology: it's also embracing a culture and a community. You 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 this chapter, you will learn about the following:

- The Node.js philosophy or the "Node way"
- The reactor pattern—the mechanism at the heart of the Node.js asynchronous event-driven architecture
- What it means to run JavaScript on the server compared to the browser

The Node.js philosophy

www.packtpub.com/support/errata, select your book, click on the **Errata Submission Form** link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you could provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Any command-line input or output is written as follows:

```
node replier.js
node requestor.js
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To explain the problem, we will create a little **web spider**, a command-line application that takes in a web URL as the input and downloads its contents locally into a file."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Most URLs are linked through our own short URL system to make it easier for readers coming through the print edition to access them. These links are in the form nodejsdp.link/some-descriptive-id.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you could report this to us. Please visit

https://static.packt-cdn.com/downloads/9781839214110_ColorImages.pdf.

Conventions used

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning:

- **Code words in text**: `server.listen(handle)`
- **Pathname**: `src/app.js`
- **Dummy URL**: `http://localhost:8080`

A block of code is generally formatted using StandardJS conventions (nodejsdp.link/standard) and it is set as follows:

```
import zmq from 'zeromq'
async function main () {
  const sink = new zmq.Pull()
  await sink.bind('tcp://*:5017')
  for await (const rawMessage of sink) {
    console.log('Message from worker: ', rawMessage.toString())
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted in bold:

```
const wss = new ws.Server({ server })
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Message: ${msg}`)
    redisPub.publish('chat_messages', msg)
  })
})
```

Download the example code files

You can download the example code files for this book from your account at www.packt.com/. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at http://www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for macOS
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at nodejsdp.link/repo. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

Revealing Constructor pattern, the *Builder* pattern, and the *Singleton* pattern.

Chapter 8, Structural Design Patterns, continues the exploration of traditional design patterns in Node.js, covering structural design patterns such as *Proxy*, *Decorator*, and *Adapter*.

Chapter 9, Behavioral Design Patterns, concludes the conversation around traditional design patterns in Node.js by introducing behavioral design patterns like *Strategy*, *State*, *Template*, *Middleware*, *Command*, and *Iterator*.

Chapter 10, Universal JavaScript for Web Applications, explores one of the most interesting capabilities of modern JavaScript web applications: being able to share code between the frontend and the backend. Throughout this chapter, you will learn the basic principles of Universal JavaScript by building a simple web application using modern tools and libraries.

Chapter 11, Advanced Recipes, takes a problem-solution approach to show you how some common coding and design intricacies can be approached with ready-to-use solutions.

Chapter 12, Scalability and Architectural Patterns, teaches you the basic techniques and patterns for scaling a Node.js application.

Chapter 13, Messaging and Integration Patterns, presents the most important messaging patterns, teaching you how to build and integrate complex distributed systems using Node.js and its ecosystem.

To get the most out of this book

To get the most out of this book you can download the example code files and the color images as per the instructions below.

What this book covers

Chapter 1, The Node.js Platform, serves as an introduction to the world of Node.js application design by showing the patterns at the core of the platform itself. It covers the Node.js ecosystem and its philosophy, and provides a quick introduction to the Node.js internals and the reactor pattern.

Chapter 2, The Module System, dives into the module systems available in Node.js, underlining the differences between CommonJS and the more modern ES modules from the ECMAScript 2015 specification.

Chapter 3, Callbacks and Events, introduces the first steps towards learning asynchronous coding and its patterns, discussing and comparing callbacks and the event emitter (observer pattern).

Chapter 4, Asynchronous Control Flow Patterns with Callbacks, introduces a set of patterns and techniques for efficiently handling asynchronous control flow in Node.js using callbacks. This chapter teaches you some traditional ways to mitigate the "callback hell" problem using plain JavaScript.

Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await, progresses with the exploration of more sophisticated and modern asynchronous control flow techniques.

Chapter 6, Coding with Streams, dives deep into one of the most important tools in Node.js: streams. It shows you how to process data with transform streams and how to combine them into different patterns.

Chapter 7, Creational Design Patterns, starts to explore the traditional design patterns in Node.js. In this chapter, you will learn about some of the most popular creational design patterns, namely the *Factory* pattern, the

Who this book is for

This book is for developers who have already had initial contact with Node.js and now want to get the most out of it in terms of productivity, design quality, and scalability. You are only required to have some prior exposure to the technology through some basic examples and some degree of familiarity with the JavaScript language, since this book will cover some basic concepts as well. Developers with intermediate experience in Node.js will also find the techniques presented in this book beneficial.

Some background in software design theory is also an advantage to understand some of the concepts presented.

This book assumes that you have a working knowledge of web application development, web services, databases, and data structures.

can scale. You will be able to apply these principles to novel problems that don't fall within the scope of existing patterns.

- **Code in "modern JavaScript"**:
  JavaScript has been around since 1995, but a lot has changed since its first inception, especially in these last few years. This book will take advantage of the most modern JavaScript features, like the class syntax, promises, generator functions, and async/await, giving you a properly up-to-date experience.

Throughout the book, you will be presented with real-life libraries and technologies, such as LevelDB, Redis, RabbitMQ, ZeroMQ, Express, and many others. They will be used to demonstrate a pattern or technique, and besides making the example more useful, these will also give you great exposure to the Node.js ecosystem and its set of solutions.

Whether you use or plan to use Node.js for your work, your side project, or for an open source project, recognizing and using well-known patterns and techniques will allow you to use a common language when sharing your code and design, and on top of that, it will help you get a better understanding of the future of Node.js and how to make your own contributions a part of it.

What you need for this book

To experiment with the code, you will need a working installation of Node.js version 14 (or greater) and npm version 6 (or greater). If some examples will require you to use some extra tooling, these will be described accordingly in place. You will also need to be familiar with the command line, know how to install an npm package, and know how to run Node.js applications. Finally, you will need a text editor to work with the code and a modern web browser.

enthusiastic and helpful community, and most importantly, its very own culture based on simplicity, pragmatism, and extreme modularity.

However, because of these peculiarities, Node.js development gives you a very different feel compared to other server-side platforms, and any developer new to this paradigm will often feel unsure about how to tackle even the most common design and coding problems effectively. Common questions include: *How do I organize my code? What's the best way to design this? How can I make my application more modular? How do I handle a set of asynchronous calls effectively? How can I make sure that my application will not collapse while it grows?* Or more simply, *what's the right way to implement this?* Fortunately, Node.js has become a mature enough platform and most of these questions can now be easily answered with a design pattern, a proven coding technique, or a recommended practice. The aim of this book is to guide you through this emerging world of patterns, techniques, and practices, showing you what the proven solutions to the most common problems are and teaching you how to use them as the starting point to building the solution to your particular problem.

By reading this book, you will learn the following:

- **The "Node way"**:
  How to use the right point of view when approaching Node.js development. You will learn, for example, how different traditional design patterns look in Node.js, or how to design modules that do only one thing.
- **A set of patterns to solve common Node.js design and coding problems**:
  You will be presented with a "Swiss Army knife" of patterns, ready to use in order to efficiently solve your everyday development and design problems.
- **How to write scalable and efficient Node.js applications**:
  You will gain an understanding of the basic building blocks and principles of writing large and well-organized Node.js applications that

Minwoo Jung is a Node.js core collaborator and works for NodeSource as a full-time software engineer. He specializes in web technologies with more than 10 years of experience and used to publish the weekly updates on the official Node.js website. When he isn't glued to a computer screen, he spends time hiking with his friends.

Contributors

About the authors

Mario Casciaro is a software engineer and entrepreneur. Since he was a child he's been in love with building things, from LEGO spaceships to programs written on his Commodore 64, his first computer. When in college, he used to work more on side projects than on assignments and he published his first open source project on SourceForge back in 2006, it was around 30,000 lines of C++ code. After graduating with a master's degree in software engineering, Mario worked at IBM for a number of years, first in Rome, then in the Dublin Software Lab. He currently splits his time between Var7 Technologies—his own software company—and his role as lead engineer at D4H Technologies where he creates software for emergency response teams. He is a big supporter of pragmatism and simplicity.

The story of this book starts with you all who are reading this book. You make all our efforts worthwhile. Thanks also to the readers who contributed to the success of the first two editions, providing invaluable feedback, writing reviews, and spreading the word about the book.

Thanks to the Packt team, who worked hard to make this book a reality; thanks to Tom Jacob, Jonathan Malysiak, Saby D'silva, Bhavesh Amin, Tushar Gupta, Kishor Rit, Joanne Lovell.

For this book, I had the honor to work with a team of top-class technical reviewers: Roberto Gambuzzi, Minwoo Jung, Kyriakos Markakis, Romina Miraballes, Peter Poliwoda, Liran Tal, and Tomas Della Vedova. Thanks for lending your expertise to make this book perfect.