Community Experience Distilled

Node.js Design Patterns

Get the best out of Node.js by mastering a series of patterns and techniques to create modular, scalable, and efficient applications

Mario Casciaro

[PACKT] PUBLISHING open source* community experience distilled

Node.js Design Patterns

Get the best out of Node.js by mastering a series of patterns and techniques to create modular, scalable, and efficient applications

Mario Casciaro

[PACKT] PUBLISHING open source
community experience distilled

BIRMINGHAM - MUMBAI

Node.js Design Patterns

First published: December 2014

Production reference: 1231214

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

ISBN 978-1-78328-731-4

www.packtpub.com

Cover image by Artie Ng (artherng@yahoo.com.au)

Credits

Author
Mario Casciaro

Reviewers
Afshin Mehrabani
Joel Purra
Alan Shaw

Commissioning Editor
Julian Ursell

Acquisition Editor
Rebecca Youé

Content Development Editor
Sriram Neelakantan

Technical Editor
Menza Mathew

Copy Editors
Shambhavi Pai
Rashmi Sawant

Project Coordinator
Aboli Ambardekar

Proofreaders
Stephen Copestake
Ameesha Green
Steve Maguire

Indexers
Hemangini Bari
Mariammal Chettiyar
Rekha Nair
Tejal Soni

Graphics
Valentina D'silva
Disha Haria
Abhinash Sahu

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Mario Casciaro is a software engineer and technical lead with a passion for open source. He began programming with a Commodore 64 when he was 12, and grew up with Pascal and Visual Basic. His programming skills evolved by experimenting with x86 assembly language, C, C++, PHP, and Java. His relentless work on side projects led him to discover JavaScript and Node.js, which quickly became his new passion.

In his professional career, he worked with IBM for several years—first in Rome and then in the Dublin Software Lab. At IBM, Mario worked on products for brands such as Tivoli, Cognos, and Collaboration Solutions, using a variety of technologies from C to PHP and Java. He then plunged into the adventurous world of start ups to work full time on Node.js projects. He ended up working in a lighthouse, at D4H Technologies, where he led the development of a real-time platform to manage emergency operations.

Acknowledgments

This book is the result of an amazing amount of work, knowledge, and perseverance from many people. A big thanks goes to the entire team at Packt who made this book a reality, from the editors to the project coordinator; in particular, I would like to thank Rebecca Youé and Sriram Neelakantan for their guidance and patience during the toughest parts of the writing process. Kudos to Alan Shaw, Joel Purra, and Afshin Mehrabani who dedicated their time and expertise to reviewing the technical content of the book; every comment and advice was really invaluable in bringing this work up to production quality. This book would not exist without the efforts of so many people who made Node.js a reality—from the big players, who continuously inspired us, to the contributor of the smallest module.

In these months, I also learned that a book is only possible with the support and encouragement of all the people around you. My gratitude goes to all my friends who heard the phrase "Today I can't, I have to work on the book" too many times; thanks to Christophe Guillou, Zbigniew Mrowinski, Ryan Gallagher, Natalia Lopez, Ruizhi Wang, and Davide Lionello for still talking to me. Thanks to the D4H crew, for their inspiration and understanding, and for giving me the chance to work on a first-class product.

Thanks to all the friends back in Italy, to the legendary company of Taverna and Centrale, to the lads of Lido Marini for always giving me a great time, laughing and having fun. I'm sorry for not being present in the past few months.

Thanks to my Mom and Dad, and to my brother and sister, for their unconditional love and support.

At last, you have to know that there is another person who wrote this book along with me, that's Miriam, my girlfriend, who walked throughout this long journey with me and supported me night and day, regardless of how difficult it was. There's nothing more one could wish for. I send all my love and gratitude to her. Many adventures await us.

About the Reviewers

Afshin Mehrabani is an open source programmer. He is studying to be a computer software engineer. He started programming and web development when he was 12 years old, and started with PHP as well. Later, he joined the Iran Technical and Vocational Training Organization. He secured the first place and received a gold medal in a competition that was conducted across the entire country in the area of web development. He became a member of Iran's National Elites Foundation after producing a variety of new programming ideas.

He was a software engineer at Tehran Stock Exchange and is now the head of the web development team in Yara International. He cofounded the Usablica team in early 2012 to develop and produce usable applications. He is the author of IntroJs, WideArea, flood.js, and other open source projects. He has contributed to Socket.IO, Engine.IO, and other open source projects. He is also interested in creating and contributing to open source applications, writing programming articles, and challenging himself with new programming technologies.

He has written different articles on JavaScript, Node.js, HTML5, and MongoDB, which have been published on different academic websites. Afshin has 5 years of experience in PHP, Python, C#, JavaScript, HTML5, and Node.js in many financial and stock-trading projects.

Joel Purra started toying around with computers some time before his teens, seeing them as another kind of a video-gaming device. It was not long before he took apart (sometimes broke and subsequently fixed) any computer he came across, in between 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 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 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 giving me 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!

Alan Shaw describes himself as a web developer who discovers the limits of the possible by venturing a little way past them into the impossible. Alan has built and styled the Web every day since graduating from the University of Bath with a degree in computer science. He is an advocate of functional programming and has worked with JavaScript for as long as he can remember.

Alan and Oli Evans own and run TABLEFLIP, a web development company that focuses on Node.js, good client relationships, and giving back to the community through open source projects.

In his spare time, Alan hacks on npm modules, browserify transforms, and grunt plugins. He builds and maintains David (`https://david-dm.org`), co-organizes the meetups for *Nodebots of London* and *Meteor London*, hacks on hardware, pilots nano copters into walls, and is a cofounder of the JavaScript Adventure Club.

Table of Contents

Preface

Node.js is considered by many as a game-changer—the biggest shift of the decade in web development. It is loved not just for its technical capabilities, but also for the change of paradigm that it introduced in web development.

First, Node.js applications are written in JavaScript, the language of the web, the only programming language supported natively by a majority of web browsers. This aspect only enables scenarios such as single-language application stacks and sharing of code between the server and the client. Node.js itself is contributing to the rise and evolution of the JavaScript language. People realize that using JavaScript on the server is not as bad as it is in the browser, and they will soon start to love it for its pragmatism and for its hybrid nature, half way between object-oriented and functional programming.

The second revolutionizing factor is its single-threaded, asynchronous architecture. Besides obvious advantages from a performance and scalability point of view, this characteristic changed the way developers approach concurrency and parallelism. Mutexes are replaced by queues, threads by callbacks and events, and synchronization by causality.

The last and most important aspect of Node.js lies in its ecosystem: the npm package manager, its constantly growing database of modules, its 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 the 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 problem 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 of doing 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 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 a Node.js design problem. 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 modular 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 and you will be able to apply these principles to novel problems that don't fall within the scope of existing patterns.

Throughout the book, you will be presented with several real-life libraries and technologies, such as LevelDb, Redis, RabbitMQ, ZMQ, 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 about the future of Node.js and how to make your own contributions a part of it.

What this book covers

Chapter 1, *Node.js Design Fundamentals*, 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 reactor pattern, the callback pattern, the module pattern, and the observer pattern.

Chapter 2, *Asynchronous Control Flow Patterns*, introduces a set of patterns and techniques for efficiently handling asynchronous control flow in Node.js. This chapter teaches you how to mitigate the "callback hell" problem using plain JavaScript, the async library, Promises, and Generators.

Chapter 3, C*oding with Streams*, dives deeply into one of the most important patterns in Node.js: Streams. It shows you how to process data with transform streams and how to combine them into different layouts.

Chapter 4, *Design Patterns*, deals with a controversial topic: traditional design patterns in Node.js. It covers the most popular conventional design patterns and shows you how unconventional they might look in Node.js.

Chapter 5, *Wiring Modules*, analyzes the different solutions for linking the modules of an application together. In this chapter, you will learn design patterns such as Dependency Injection and Service locator.

Chapter 6, *Recipes*, takes a problem-solution approach to show you how some common coding and design challenges can be solved with ready-to-use solutions.

Chapter 7, *Scalability and Architectural Patterns,* teaches you the basic techniques and patterns for scaling a Node.js application.

Chapter 8, *Messaging and Integration Patterns*, presents the most important messaging patterns, teaching you how to build and integrate complex distributed systems using ZMQ and AMQP.

What you need for this book

To experiment with the code, you will need a working installation of Node.js version 0.10 (or greater) and npm. Some examples will require Node.js 0.11 or greater. You will also need to be familiar with the command prompt, know how to install an npm package, and know how to run Node.js applications. You will also need a text editor to work with the code and a web browser.

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, 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, JavaScript, web services, databases, and data structures.

Conventions

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

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
var zmq  = require('zmq')
var sink = zmq.socket('pull');
sink.bindSync("tcp://*:5001");

sink.on('message', function(buffer) {
  console.log('Message from worker: ', buffer.toString());
});
```

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

```
function produce() {
  [...]
  variationsStream(alphabet, maxLength)
    .on('data', function(combination) {
        [...]
```

```
            var msg = {searchHash: searchHash, variations: batch};
            channel.sendToQueue('jobs_queue',
              new Buffer(JSON.stringify(msg)));
            [...]
          }
        })
      [...]
    }
```

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

```
node replier
node requestor
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, 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 in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Node.js Design Fundamentals

Some principles and design patterns literally define the Node.js platform and its ecosystem; the most peculiar ones are probably its asynchronous nature and its programming style that makes heavy use of callbacks. However, there are other fundamental components that characterize the platform; for example, its module system, which allows multiple versions of the same dependency to coexist in an application, and the observer pattern, implemented by the `EventEmitter` class, which perfectly complements callbacks when dealing with asynchronous code. It's therefore 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 this chapter, we will learn the following topics:

- The Node.js philosophy, the "Node way"
- The reactor pattern: the mechanism at the heart of the Node.js asynchronous architecture
- The Node.js callback pattern and its set of conventions
- The module system and its patterns: the fundamental mechanisms for organizing code in Node.js
- The observer pattern and its Node.js incarnation: the `EventEmitter` class

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 in 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 functionality, 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 fast 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 functionality 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 *module* as a fundamental mean to structure the code of a program. It is the brick for creating applications and reusable libraries called *packages* (a package is also frequently referred to as just 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 solving 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 in 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 at 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 only a minimal set of functionality. The main advantage here is an 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 interested only 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 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. Simplicity is the most important consideration in a design.*"

Designing a simple, as opposed to a perfect, feature-full software, is a good practice for several reasons: it takes less effort to implement, allows faster shipping with less resources, is easier to adapt, and is easier to maintain and understand. These factors foster the 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 the reality and we would probably have more success in trying to get something working sooner and with reasonable complexity, instead of trying to create a near-perfect software with a 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.

The reactor pattern

In this section, we will analyze the reactor pattern, which is the heart of the Node.js asynchronous nature. 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 don't depend only on technical aspects, and they can be many orders of magnitude slower than the disk or network.

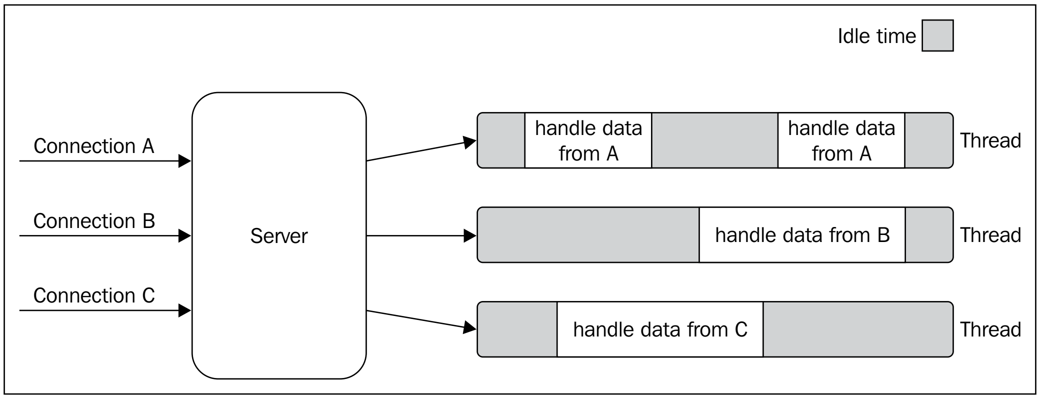
Blocking I/O

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 case of a 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 read 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 handle 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 blocks 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 a 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
    var 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 consume precious CPU only 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:

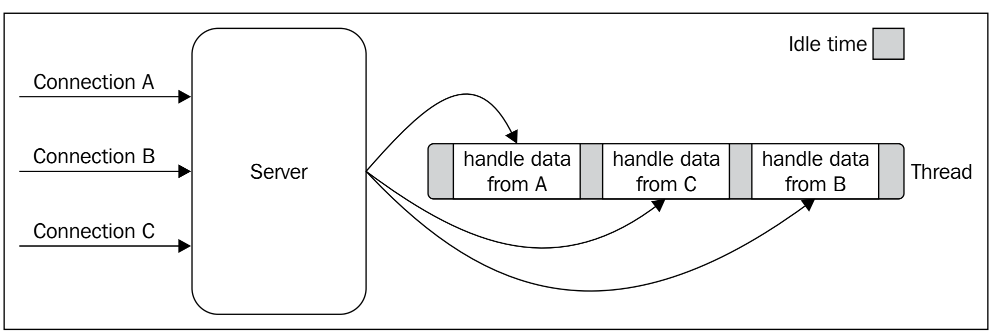
```
socketA, pipeB;
watchedList.add(socketA, FOR_READ);        //[1]
watchedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(watchedList)) {    //[2]
  //event loop
```

```
    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 a 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 is ready for a 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:

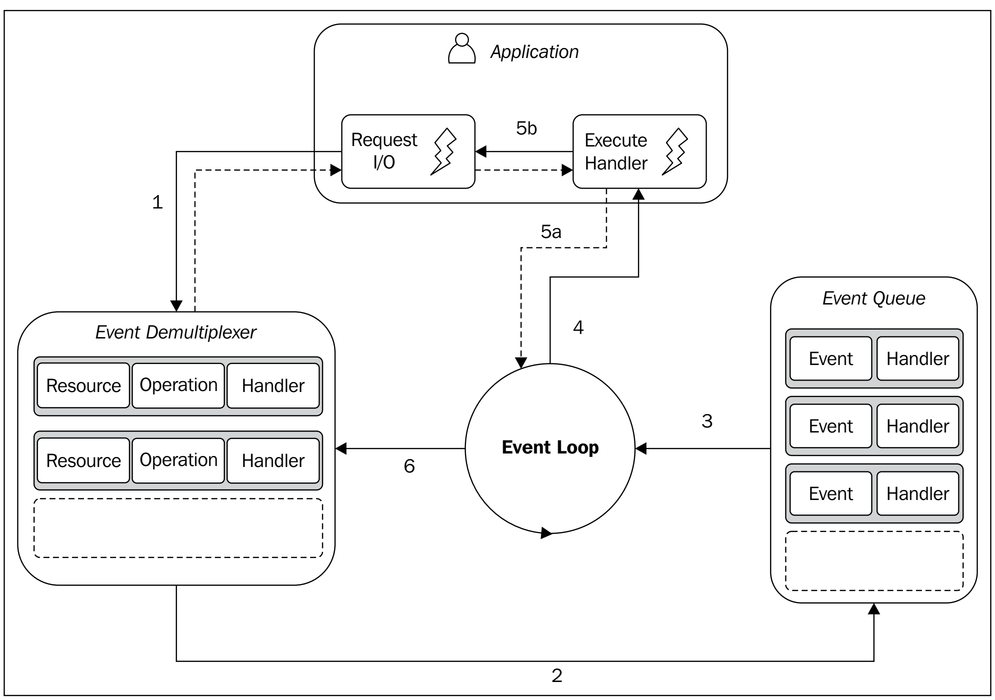


The previous image helps us understand how concurrency works in a single-threaded application using a synchronous event demultiplexer and non-blocking I/O. We can see that using only one thread does not impair our ability to run multiple I/O bound tasks *concurrently*. The tasks are spread over time, 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.

The reactor pattern

We can now introduce the **reactor pattern**, which is a specialization of the algorithm 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 the control back 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 the 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 the 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.

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 reacting 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 **I/O Completion Port API** (**IOCP**) 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 a 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 tasks.

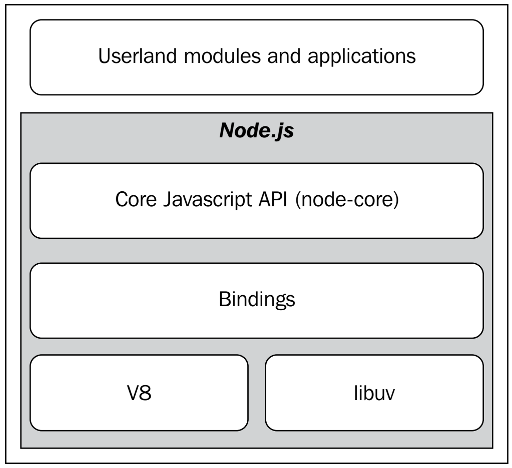
> A great resource to learn more about `libuv` is the free online book created by Nikhil Marathe, which is available at `http://nikhilm.github.io/uvbook/`.

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:



The callback pattern

Callbacks are the materialization of the handlers of the reactor pattern and they are literally one of those imprints that give Node.js its distinctive programming style. Callbacks are functions that are invoked to propagate the result of an operation and this is exactly what we need when dealing with asynchronous operations. They practically replace the use of the `return` instruction that, as we know, always executes synchronously. JavaScript is a great language to represent callbacks, because as we know, functions are first class objects and can be easily assigned to variables, passed as arguments, returned from another function invocation, or stored into data structures. Also, **closures** are an ideal construct for implementing callbacks. With closures, we can in fact reference the environment in which a function was created, practically, we can always maintain the context in which the asynchronous operation was requested, no matter when or where its callback is invoked.

> If you need to refresh your knowledge about closures, you can refer to the article on the Mozilla Developer Network at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures`.

In this section, we will analyze this particular style of programming made of callbacks instead of the `return` instructions.

The continuation-passing style

In JavaScript, a callback is a function that is passed as an argument to another function and is invoked with the result when the operation completes. In functional programming, this way of propagating the result is called **continuation-passing style**, for brevity, **CPS.** It is a general concept, and it is not always associated with asynchronous operations. In fact, it simply indicates that a result is propagated by passing it to another function (the callback), instead of directly returning it to the caller.

Synchronous continuation-passing style

To clarify the concept, let's take a look at a simple synchronous function:

```
function add(a, b) {
  return a + b;
}
```

There is nothing special here; the result is passed back to the caller using the `return` instruction; this is also called **direct style**, and it represents the most common way of returning a result in synchronous programming. The equivalent continuation-passing style of the preceding function would be as follows:

```
function add(a, b, callback) {
  callback(a + b);
}
```

The `add()` function is a synchronous CPS function, which means that it will return a value only when the callback completes its execution. The following code demonstrates this statement:

```
console.log('before');
add(1, 2, function(result) {
  console.log('Result: ' + result);
});
console.log('after');
```

Since `add()` is synchronous, the previous code will trivially print the following:

```
before
Result: 3
after
```

Asynchronous continuation-passing style

Now, let's consider the case where the `add()` function is asynchronous, which is as follows:

```
function addAsync(a, b, callback) {
  setTimeout(function() {
    callback(a + b);
  }, 100);
}
```

In the previous code, we simply use `setTimeout()` to simulate an asynchronous invocation of the callback. Now, let's try to use this function and see how the order of the operations changes:

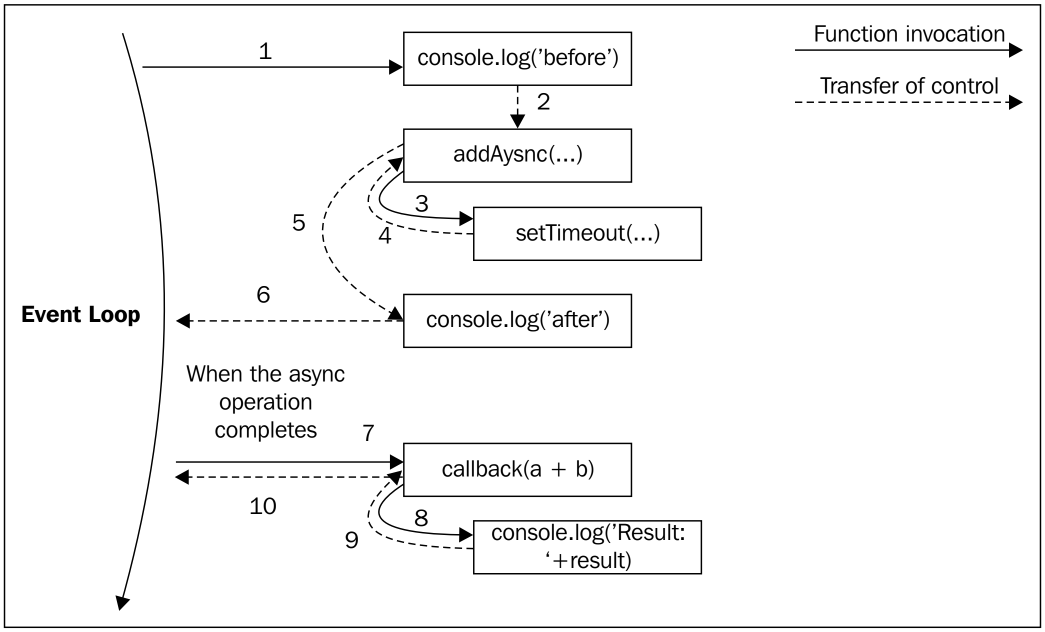
```
console.log('before');
addAsync(1, 2, function(result) {
  console.log('Result: ' + result);
});
console.log('after');
```

The preceding code will print the following:

```
before
after
Result: 3
```

Since `setTimeout()` triggers an asynchronous operation, it will not wait anymore for the callback to be executed, but instead, it returns immediately giving the control back to `addAsync()`, and then back to its caller. This property in Node.js is crucial, as it allows the stack to unwind, and the control to be given back to the event loop as soon as an asynchronous request is sent, thus allowing a new event from the queue to be processed.

The following image shows how this works:



When the asynchronous operation completes, the execution is then resumed starting from the callback provided to the asynchronous function that caused the unwinding. The execution will start from the Event Loop, so it will have a fresh stack. This is where JavaScript comes in really handy, in fact, thanks to closures it is trivial to maintain the context of the caller of the asynchronous function, even if the callback is invoked at a different point in time and from a different location.

> A **synchronous** function blocks until it completes its operations. An **asynchronous** function returns immediately and the result is passed to a handler (in our case, a callback) at a later cycle of the event loop.

Non continuation-passing style callbacks

There are several circumstances in which the presence of a callback argument might make you think that a function is asynchronous or is using a continuation-passing style; that's not always true, let's take, for example, the `map()` method of the `Array` object:

```
var result = [1, 5, 7].map(function(element) {
  return element - 1;
});
```

Clearly, the callback is just used to iterate over the elements of the array, and not to pass the result of the operation. In fact, the result is returned synchronously using a direct style. The intent of a callback is usually clearly stated in the documentation of the API.

Synchronous or asynchronous?

We have seen how the order of the instructions changes radically depending on the nature of a function - synchronous or asynchronous. This has strong repercussions on the flow of the entire application, both in correctness and efficiency. The following is an analysis of these two paradigms and their pitfalls. In general, what must be avoided, is creating inconsistency and confusion around the nature of an API, as doing so can lead to a set of problems which might be very hard to detect and reproduce. To drive our analysis, we will take as example the case of an inconsistently asynchronous function.

An unpredictable function

One of the most dangerous situations is to have an API that behaves synchronously under certain conditions and asynchronously under others. Let's take the following code as an example:

```
var fs = require('fs');
var cache = {};
function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    //invoked synchronously
    callback(cache[filename]);
  } else {
    //asynchronous function
```

```
      fs.readFile(filename, 'utf8', function(err, data) {
        cache[filename] = data;
        callback(data);
      });
    }
  }
```

The preceding function uses the `cache` variable to store the results of different file read operations. Please bear in mind that this is just an example, it does not have error management, and the caching logic itself is suboptimal. Besides this, the preceding function is dangerous because it behaves asynchronously until the cache is not set—which is until the `fs.readFile()` function returns its results—but it will also be synchronous for all the subsequent requests for a file already in the cache—triggering an immediate invocation of the callback.

Unleashing Zalgo

Now, let's see how the use of an unpredictable function, such as the one that we defined previously, can easily break an application. Consider the following code:

```
function createFileReader(filename) {
  var listeners = [];
  inconsistentRead(filename, function(value) {
    listeners.forEach(function(listener) {
      listener(value);
    });
  });

  return {
    onDataReady: function(listener) {
      listeners.push(listener);
    }
  };
}
```

When the preceding function is invoked, it creates a new object that acts as a notifier, allowing to set multiple listeners for a file read operation. All the listeners will be invoked at once when the read operation completes and the data is available. The preceding function uses our `inconsistentRead()` function to implement this functionality. Let's now try to use the `createFileReader()` function:

```
var reader1 = createFileReader('data.txt');
reader1.onDataReady(function(data) {
  console.log('First call data: ' + data);
```

```
    //...sometime later we try to read again from
    //the same file
    var reader2 = createFileReader('data.txt');
    reader2.onDataReady(function(data) {
      console.log('Second call data: ' + data);
    });
  });
```

The preceding code will print the following output:

```
First call data: some data
```

As you can see, the callback of the second operation is never invoked. Let's see why:

- During the creation of `reader1`, our `inconsistentRead()` function behaves asynchronously, because there is no cached result available. Therefore, we have all the time to register our listener, as it will be invoked later in another cycle of the event loop, when the read operation completes.
- Then, `reader2` is created in a cycle of the event loop in which the cache for the requested file already exists. In this case, the inner call to `inconsistentRead()` will be synchronous. So, its callback will be invoked immediately, which means that also all the listeners of `reader2` will be invoked synchronously. However, we are registering the listeners after the creation of `reader2`, so they will never be invoked.

The callback behavior of our `inconsistentRead()` function is really unpredictable, as it depends on many factors, such as the frequency of its invocation, the filename passed as argument, and the amount of time taken to load the file.

The bug that we've just seen might be extremely complicated to identify and reproduce in a real application. Imagine to use a similar function in a web server, where there can be multiple concurrent requests; imagine seeing some of those requests hanging, without any apparent reason and without any error being logged. This definitely falls under the category of *nasty* defects.

Isaac Z. Schlueter, creator of `npm` and former Node.js project lead, in one of his blog posts compared the use of this type of unpredictable functions to *unleashing Zalgo*. If you're not familiar with Zalgo, you are invited to find out what it is.

You can find the original Isaac Z. Schlueter's post at `http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony`.

Using synchronous APIs

The lesson to learn from the unleashing Zalgo example is that it is imperative for an API to clearly define its nature, either synchronous or asynchronous.

One suitable fix for our `inconsistentRead()` function, is to make it totally synchronous. This is possible because Node.js provides a set of synchronous direct style APIs for most of the basic I/O operations. For example, we can use the `fs.readFileSync()` function in place of its asynchronous counterpart. The code would now be as follows:

```
var fs = require('fs');
var cache = {};
function consistentReadSync(filename) {
  if(cache[filename]) {
    return cache[filename];
  } else {
    cache[filename] = fs.readFileSync(filename, 'utf8');
    return cache[filename];
  }
}
```

We can see that the entire function was also converted to a direct style. There is no reason for the function to have a continuation-passing style if it is synchronous. In fact, we can state that it is always a good practice to implement a synchronous API using a direct style; this will eliminate any confusion around its nature and will also be more efficient from a performance perspective.

Pattern: prefer the direct style for purely synchronous functions.

Please bear in mind that changing an API from CPS to a direct style, or from asynchronous to synchronous, or vice versa might also require a change to the style of all the code using it. For example, in our case, we will have to totally change the interface of our `createFileReader()` API and adapt it to work always synchronously.

Also, using a synchronous API instead of an asynchronous one has some caveats:

- A synchronous API might not be always available for the needed functionality.
- A synchronous API will block the event loop and put the concurrent requests on hold. It practically breaks the Node.js concurrency, slowing down the whole application. We will see later in the book what this really means for our applications.

In our `consistentReadSync()` function, the risk of blocking the event loop is partially mitigated, because the synchronous I/O API is invoked only once per each filename, while the cached value will be used for all the subsequent invocations. If we have a limited number of static files, then using `consistentReadSync()` won't have a big effect on our event loop. Things can change quickly if we have to read many files and only once. Using synchronous I/O in Node.js is strongly discouraged in many circumstances; however, in some situations, this might be the easiest and most efficient solution. Always evaluate your specific use case in order to choose the right alternative.

> Use blocking API only when they don't affect the ability of the application to serve concurrent requests.

Deferred execution

Another alternative for fixing our `inconsistentRead()` function is to make it purely asynchronous. The trick here is to schedule the synchronous callback invocation to be executed "in the future" instead of being run immediately in the same event loop cycle. In Node.js, this is possible using `process.nextTick()`, which defers the execution of a function until the next pass of the event loop. Its functioning is very simple; it takes a callback as an argument and pushes it on the top of the event queue, in front of any pending I/O event, and returns immediately. The callback will then be invoked as soon as the event loop runs again.

Let's apply this technique to fix our `inconsistentRead()` function as follows:

```
var fs = require('fs');
var cache = {};
function consistentReadAsync(filename, callback) {
  if(cache[filename]) {
    process.nextTick(function() {
      callback(cache[filename]);
    });
  } else {
    //asynchronous function
    fs.readFile(filename, 'utf8', function(err, data) {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

Now, our function is guaranteed to invoke its callback asynchronously, under any circumstances.

Another API for deferring the execution of code is `setImmediate()`, which—despite the name—might actually be *slower* than `process.nextTick()`. While their purpose is very similar, their semantic is quite different. Callbacks deferred with `process.nextTick()` run before any other I/O event is fired, while with `setImmediate()`, the execution is queued behind any I/O event that is already in the queue. Since `process.nextTick()` runs before any already scheduled I/O, it might cause I/O starvation under certain circumstances, for example, a recursive invocation; this can never happen with `setImmediate()`. We will learn to appreciate the difference between these two APIs when we analyze the use of deferred invocation for running synchronous CPU-bound tasks later in the book.

> Pattern: we guarantee that a callback is invoked asynchronously by deferring its execution using `process.nextTick()`.