EXPERT INSIGHT

Early Access

Node.js Design Patterns

Level up your Node.js skills and design production-grade applications using proven techniques

Fourth Edition

Mario Casciaro
Luciano Mammino

‹packt›

Node.js Design Patterns

Early Access Publication: Node.js Design Patterns

Early Access Production Reference: B18437

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN: 978-1-80323-894-4

www.packt.com

Table of Contents

Node.js Design Patterns, Fourth Edition: Level up your Node.js skills and design production-grade applications using proven techniques

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1

The Node.js Platform

Over the past decade, Node.js has become a cornerstone of modern web development, revolutionizing how companies of all sizes approach building scalable and high-performance applications. Its ability to handle asynchronous operations allows for real-time, responsive solutions that can manage great amounts of concurrent connections efficiently. Node.js offers a unified environment where JavaScript can run seamlessly on both the client and the server, streamlining the development process.Mastering Node.js goes beyond learning its syntax and libraries; in fact, some key ideas and patterns shape how developers use Node.js and its ecosystem. Its asynchronous nature means that we need to embrace asynchronous primitives such as callbacks, promises, and async/await, and those come with their own unique challenges and patterns. In this first chapter, we'll look into why Node.js works this way. This isn't just theoretical: knowing how Node.js works at its core will give you a strong foundation for understanding the reasoning behind the more complex topics and patterns that we will cover later in the book.Another defining aspect of Node.js is its opinionated philosophy. Embracing Node.js means joining a culture and community that deeply influence how we design applications and interact with the broader ecosystem.In this chapter, you’ll discover:

- 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

To highlight the importance of this philosophy, let me share a bit of my (Luciano's) journey with Node.js. As a web developer, I had a solid background in frontend JavaScript (yes, lots of jQuery!), so when I discovered Node.js—version 0.12 at the time—I was excited to use JavaScript on the backend and replace the other languages I was working with, like PHP, Java, and .NET.

I started learning Node.js with the first edition of this very book, which Mario had authored solo. After finishing the book, I was eager to build something, so I took on a personal project: downloading an entire photo gallery from Flickr (a popular photo hosting site at the time). Flickr didn't offer a way to download all the photos in one go, so I decided to use its API and my new Node.js skills to create a CLI tool that could do just that.

This project was the perfect opportunity to leverage Node.js's asynchronous nature. Downloading hundreds of files from URLs is ideal for concurrency—there's no reason to fetch them one by one when you can download several at once. However,

doing this effectively required limiting concurrency to avoid overwhelming system resources like memory and network bandwidth. After building the tool, I shared the project on GitHub (`nodejsdp.link/flickr-set-get` *) and sought feedback from various Node.js communities. And I got a lot of feedback! Many people pointed out that, while my code worked, it didn't fully embrace the "Node way." My approach still had traces of a PHP-like style, which made it harder to integrate smoothly with the broader Node.js ecosystem.*

That feedback was invaluable—it helped me improve my Node.js skills and understand the importance of adopting its design principles. And here's a fun twist: Mario was one of the people who gave me feedback on that project! That interaction sparked a connection between us, which eventually led to me joining him as a co-author for the second edition of the book a few years later.

So, maybe there's a lesson in not being afraid to share your work and ask for candid feedback. The Node.js community is quite supportive and helpful, providing countless opportunities to learn and grow. Who knows where those opportunities might lead!

Enough with the motivational story—let's dive into some learning.

The Node.js philosophy

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.These guidelines aren't strict rules and should be used with pragmatism. However, they can be very helpful when you're looking for inspiration in designing your software.

If you are curious to see other examples of software development philosophies, you can find an extensive list on Wikipedia at `nodejsdp.link/dev-philosophies`.

Small core

Historically, the Node.js core, which includes the runtime and built-in modules, has been kept minimal, with most features left to the "**userland**" (or **userspace**)—the ecosystem of modules outside the core. This approach has allowed the community to experiment and develop new solutions quickly, rather than relying on a single, slowly evolving core solution. Keeping the core minimal makes it easier to maintain and positively impacts the community by encouraging innovation in the userland modules. In recent years, the principle of keeping the Node.js core minimal has become less strict. The community has shown interest in having more built-in features, so several important capabilities have been added directly to the core. These include command-line argument parsing, WebSockets, a unit testing framework, file watch capability, file globbing, the web `fetch` API, and more. This shift doesn't change the core principles of the Node.js community but reflects its evolution. Many common interfaces have become mature and stable, making it sensible to include them in the core for easy access without needing third-party libraries.

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 also, 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, pnpm,** 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.For example, suppose we have a project that uses two dependencies: `depA` and `depB`. Both `depA` and `depB` rely on a third library, `depC`, but they need different versions— `depA` requires `depC@1.0.0`, while `depB` requires `depC@2.0.0`. Node.js handles this without conflict by organizing the dependencies like this:

```
.
└── node_modules
    ├── depA@1.0.0
    │   └── node_modules
    │       └── depC@1.0.0
    └── depB@1.0.0
```

```
            └── node_modules
                └── depC@2.0.0
```

Here, two versions of `depC` are installed: one under `depA` for version `1.0.0` and another under `depB` for version `2.0.0`. This way, Node.js ensures that when `depA` needs `depC`, it loads version `1.0.0`, and when `depB` needs `depC`, it loads version `2.0.0`, avoiding version conflicts entirely.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 few lines of code—for example, `is-sorted`, a library to check if an array is sorted (`nodejsdp.link/is-sorted`).Besides the clear advantage in terms of reusability, a small module is also:

- Easier to understand and use
- Simpler to test and maintain
- Lightweight in terms of kilobytes, ideal in the browser (since the npm registry is used by both Node.js and frontend applications), and in serverless environments that require quick start times such as AWS Lambda

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.

While the principle of small modules applies, the recent rise in supply chain vulnerabilities (`https://nodejsdp.link/supply-chain`) has made the software industry more cautious about adding third-party dependencies. This is true for Node.js projects as well. It's important to carefully evaluate if a third-party module is well-maintained and if adding a new dependency is necessary. The more third-party dependencies, the higher the risk of one of them getting compromised and affecting the project's security.

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 that they are designed to be used, not extended. Locking down the internals of a module by preventing extensions might seem inflexible, but it simplifies implementation, makes maintenance easier,

and improves usability. In practice, this means preferring to expose functions instead of classes and ensuring no internals are exposed 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. 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.
- 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, this principle

is supported by JavaScript's pragmatic nature. Instead of using complex class hierarchies, it's common to see simple classes, functions, and closures. Pure object-oriented designs often attempt to model the real world using mathematical concepts, which can overlook the real world's imperfections and complexities. In reality, software is always an approximation of reality. We are likely to be more successful if we focus on getting something functional quickly with manageable complexity, rather than striving for nearly perfect software abstractions that require extensive effort and tons of code to maintain.Throughout this book, you'll see this principle applied often. For example, traditional design patterns like Singleton or Decorator can be implemented simply, which might not be perfect but is usually practical.In Node.js, we prefer straightforward and practical solutions over complex, flawless designs. This doesn't mean we're lowering our standards; rather, we carefully weigh the trade-offs between extensive coverage and complexity versus simplicity and clear boundaries.Next, we will take a look inside the Node.js core to reveal its internal patterns and event-driven architecture.

How Node.js works

In this section, you'll learn how Node.js operates internally and be introduced to the reactor pattern, which is central to Node.js' asynchronous nature. We'll cover key concepts like the single-threaded architecture and non-blocking I/O, and show how these elements form the basis of the Node.js

platform. Understanding how Node.js works will be crucial for mastering the runtime and writing clean, efficient code.

> *While we commonly describe Node.js as "single-threaded" due to its ability to handle asynchronous tasks concurrently on a single thread, this doesn't mean it can't leverage background threads for certain operations. Node.js uses a single thread for the event loop, but when needed, it can execute CPU-intensive tasks on separate threads, allowing for more efficient handling of complex operations. In Chapter 11, Advanced Recipes, we'll explore how to use worker threads to perform such CPU-heavy tasks in parallel, demonstrating how Node.js can go beyond its single-threaded nature when necessary.*

I/O is often the bottleneck

Regardless of your choice of programming language, 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 (10 E-9 seconds), while accessing data on the disk or the network is in the order of milliseconds (10 E-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 don't only depend on technical aspects, and it can be many orders of magnitude slower than the disk or network.Despite the inherent slowness of I/O, Node.js is designed to handle it with remarkable efficiency. Its non-blocking, event-driven architecture allows the system to remain responsive even while waiting for I/O operations to complete, making it an excellent choice for applications that need to perform large amounts of I/O without sacrificing performance. But to understand how the non-blocking model of Node.js works, we need to briefly explore the concept of blocking I/O.

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
print(data)
```

It is really important to understand that a web server that is implemented using blocking I/O will not be able to handle multiple connections in the same thread. This is because each I/O operation on a socket will block the processing of any other connection. The traditional approach to solving this problem is to use a separate thread (or process) to handle each concurrent connection. This way, a thread blocked on an I/O operation will not impact the availability of the other connections because they are handled in separate threads.The following diagram illustrates this scenario:

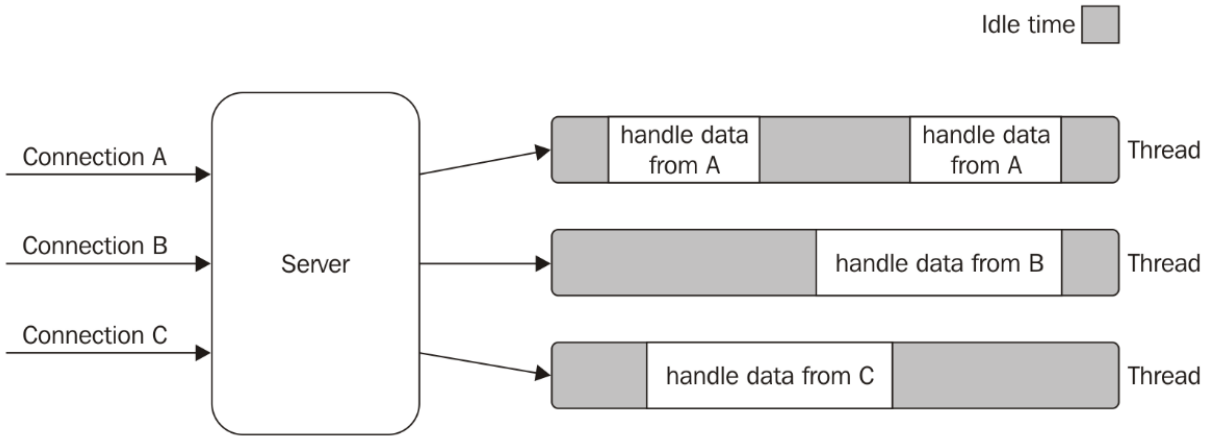

Figure 1.1: Using multiple threads to process multiple connections

Figure 1.1 lays emphasis on the amount of time each thread is idle and 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 will soon realize how many times a thread must 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 means wasting precious memory and CPU cycles.The key takeaway here is that achieving concurrency with a traditional blocking approach typically requires multiple threads, which are costly in terms of system resources. In contrast, Node.js uses a non-blocking, single-threaded approach that can be far more efficient when handling I/O operations.

> *For a deeper dive into the concepts discussed, we recommend checking out this Wikipedia page on computing threads:* `nodejsdp.link/thread`.

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 modify an existing file descriptor (which is a reference to a local file or network socket) to change its operating mode to non-blocking using the `O_NONBLOCK` flag. When a resource is in non-blocking mode, any read operation will return the error code `EAGAIN` if there is no data ready to be read.The simplest way to han-

dle non-blocking I/O is to actively check the resource in a loop until data is available, a method known as **busy-waiting**. The following pseudocode demonstrates how to read from multiple resources using non-blocking I/O and an active polling loop:

```
resources = [socketA, socketB, fileA]
while (!resources.isEmpty()) {
  for (resource of resources) {
    // try to read
    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
      resources.remove(i)
    } else {
      //some data was received, process it
      consumeData(data)
    }
  }
}
```

As you can see, with this simple technique, it is 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 cycles for iterating over

resources that are unavailable most of the time. Polling algorithms usually result in a huge amount of wasted CPU time. Let's see how we can implement non-blocking I/O in a more efficient way.

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. We are talking about the **synchronous event demultiplexer** (also known as the **event notification interface**).

> *If you are unfamiliar with the term, in telecommunications,* **multiplexing** *refers to the method by which multiple signals are combined into one so that they can be easily transmitted over a medium with limited capacity.*
>
> > **Demultiplexing** *refers to the opposite operation, whereby the signal is split again into its original components. Both terms are used in other areas (for example, video processing) to describe the general operation of combining different things into one, and vice versa.*

This type of event demultiplexer monitors multiple resources and generates an event (or a set of events) when a read or write operation on one of those

resources completes. The advantage is that the synchronous event demultiplexer blocks until there are new events to process. The following pseudocode demonstrates an algorithm using a generic synchronous event demultiplexer to read from two different resources:

```
watchList.add(socketA, FOR_READ) // (1)
watchList.add(fileB, FOR_READ)
while (events = demultiplexer.watch(watchList)) {
  // event loop
  for (event of events) { // (3)
    // This read will never block and will always
    data = event.resource.read()
    if (data === RESOURCE_CLOSED) {
      // the resource was closed, remove it from
      demultiplexer.unwatch(event.resource)
    } else {
      // some actual data was received, process i
      consumeData(data)
    }
  }
}
```

Let's see what happens in 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 demultiplexer is set up with the group of resources to be watched. The call to `demultiplexer.watch()` 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 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 the busy-waiting technique. It should now be clear why we are talking about demultiplexing; using just a single thread, we can deal with multiple resources. *Figure 1.2* will help you visualize what's happening in a web server that uses a synchronous event demultiplexer and a single thread to handle multiple concurrent connections:

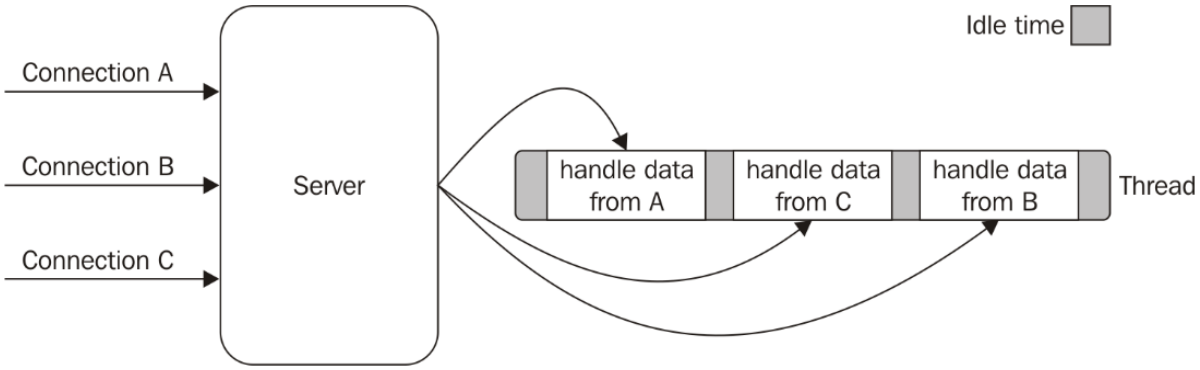

Figure 1.2: Using a single thread to process multiple connections

As this shows, 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 is clearly shown in *Figure 1.2*.But this is not the only reason for choosing this I/O model. In fact, having a single thread also has a beneficial impact on the way programmers approach concurrency in general. Throughout the book, you will see how the absence of in-process race conditions and multiple threads to synchronize allows us to use much simpler concurrency strategies.

The reactor pattern

We can now introduce the reactor pattern, which is a variation of the algorithms presented in the previous sections and what Node.js utilizes under the hood. The main idea behind the reactor pattern is to have a handler associated with each I/O operation. A handler in Node.js is represented by a

`callback` (or `cb` for short) function.The handler 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 *Figure 1.3*:

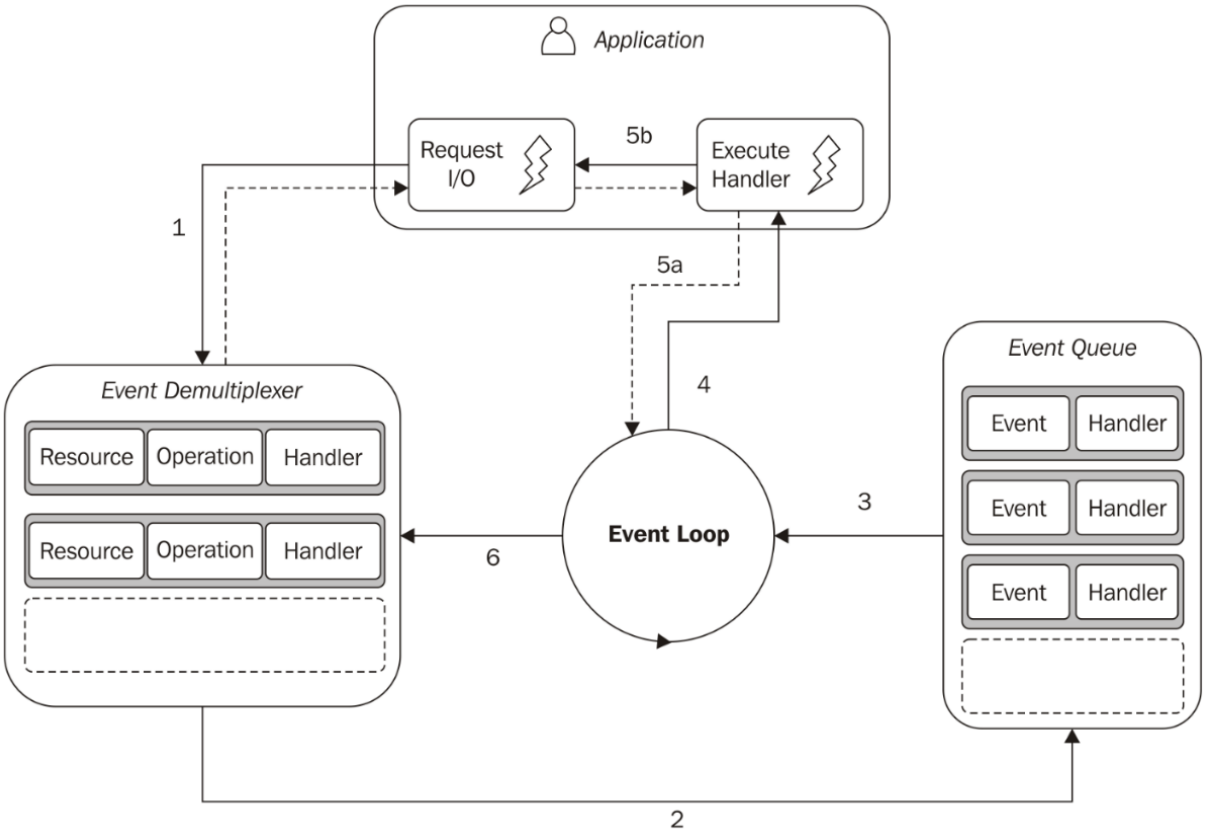

Figure 1.3: The reactor pattern

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.

1. When a set of I/O operations completes, the **Event Demultiplexer** pushes a set of corresponding events into the **Event Queue**.
2. At this point, the **Event Loop** iterates over the items of the **Event Queue**.
3. For each event, the associated handler is invoked.
4. The handler, which is part of the application code, gives back control to the **Event Loop** when its execution completes (**5 a**). While the handler executes, it can request new asynchronous operations (**5 b**), causing new items to be added to the **Event Demultiplexer** (**1**).
5. When all the items in the **Event Queue** are processed, the **Event Loop** blocks again on the **Event Demultiplexer**, which then triggers another cycle when a new event is available.

The asynchronous behavior has now become clear. The application expresses interest in accessing 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 exits when the event loop determines that there are no pending operations or events left to handle. Specifically, when there are no more active handles or requests (such as timers, open sockets, or filesystem operations) within the event demultiplexer, the event loop stops. However, certain resources like an active HTTP

server, open network sockets, or pending I/O operations keep the event loop alive by continuously registering events. For example, in the case of an HTTP server, the `http.createServer()` *call creates a server that listens on a specified port. This listening socket is considered an active handle by the event loop. As long as the server is listening, Node.js maintains this socket to accept incoming connections, which keeps the process running. The event loop will only exit when this socket is explicitly closed, and no other active handles remain in the system.*

We can finally define the reactor pattern, the pattern at the heart of Node.js: it 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 reactor pattern should not be confused with the **proactor pattern**, *even though both aim to manage multiple I/O operations concurrently. While they share a similar goal, they differ significantly in how they handle event processing. In the reactor pattern, the application has more control over when and how I/O operations are performed, as it responds to signals indicating that I/O is ready. In contrast, the proactor pattern abstracts the entire I/O process, notifying the application only once the operation is complete. Since Node.js uses the reactor pattern, we won't be diving into the proactor ap-*

proach here. However, if you're interested, you can learn more by visiting `nodejsdp.link/proactor`.

libuv, the I/O engine of Node.js

Each operating system has its own interface for the event demultiplexer: `epoll` on Linux, `kqueue` on macOS, and the I/O completion port (IOCP) API on Windows. On top of that, each I/O operation can behave quite differently depending on the type of resource, even within the same operating system. In Unix operating systems, for example, 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 native library called **libuv**, with the objective of making Node.js compatible with all the major operating systems and normalizing the non-blocking behavior of the different types of resources. libuv represents the low-level I/O engine of Node.js and is probably the most important component that Node.js is built on.Other than 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 `nodejsdp.link/uvbook`.

The complete recipe for Node.js

With our understanding of the reactor pattern and libuv, we've uncovered some of the key components that make Node.js work. However, to complete the full recipe for building the Node.js platform, we need three more crucial ingredients:

- A set of bindings responsible for wrapping and exposing libuv and other low-level functionalities 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 its efficient memory management.
- A core JavaScript library that implements the high-level Node.js API.

This is the recipe for creating Node.js, and the following image represents its final architecture:

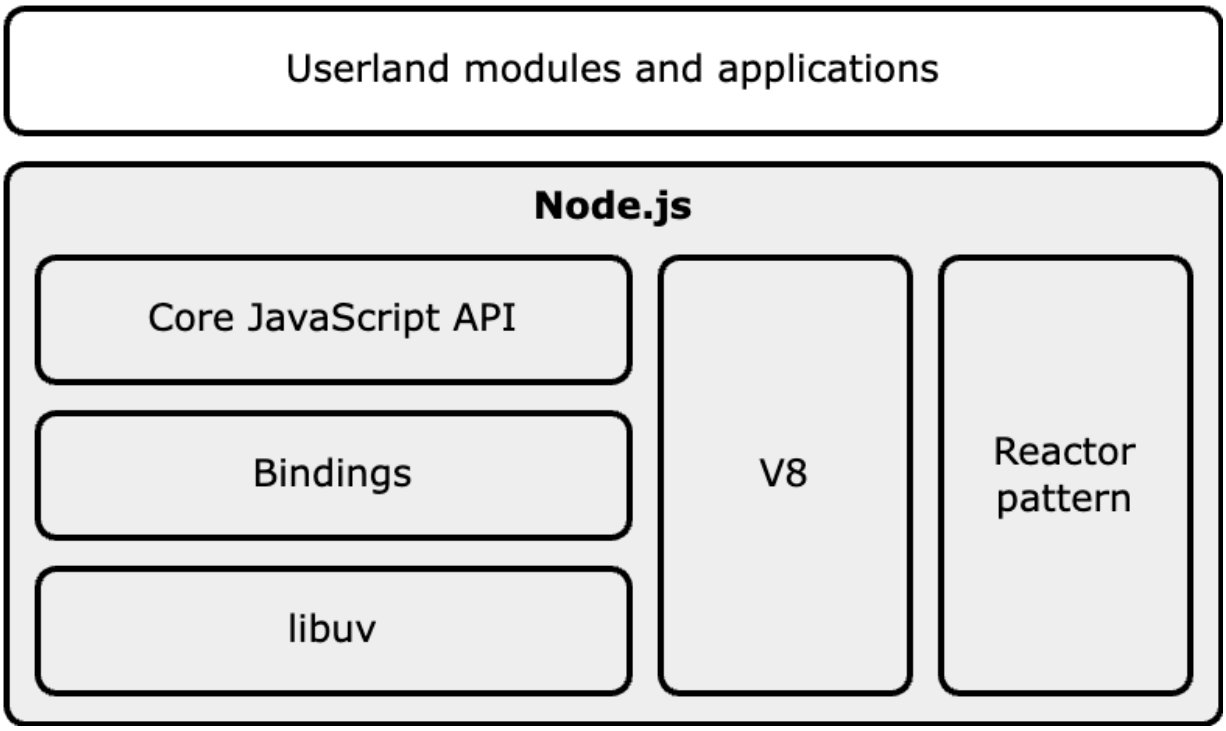

Figure 1.4: The Node.js internal components

This concludes our journey through the internal mechanisms of Node.js. Next, we'll take a look at some important aspects to take into consideration when working with JavaScript in Node.js.

JavaScript in Node.js

One important consequence of the architecture we have just analyzed is that the JavaScript we use in Node.js is somewhat different from the JavaScript we use in the browser.The major difference between running JavaScript in the browser (client-side) and in Node.js (server-side) lies in their execution environments. In the browser, JavaScript runs when a user visits a web

page, requiring a secure environment to prevent unrestricted access to the user's system and potential vulnerabilities. On the server side, however, we have more control and typically need access to databases, the filesystem, the network, and other system resources to build functional applications. As a result, Node.js has access to all the services provided by the operating system.Although both Node.js and the browser can execute JavaScript, their different use cases and security requirements lead to significantly different APIs. The most important difference is that, in Node.js, we don't have a DOM, and there is no `window` or `document` object.In this overview, we'll take a look at some key facts to keep in mind when using JavaScript in Node.js.

Run the latest JavaScript with confidence

One of the main challenges of using JavaScript in the browser is that our code needs to run on various devices and browsers. Different browsers can have small differences and may not support the latest features of the language or the web platform. This used to be a big problem in the past. If you've ever had to support Internet Explorer 6, you know what we mean. Luckily, today’s browsers are much better at following standards and they get updated frequently, which greatly reduces unexpected issues. If you want to use the newest features, you can often use *transpilers* and *polyfills* to make sure they work. However, these tools add complexity to your project, and not everything can be polyfilled.

> *A* **transpiler** *is a tool that converts code from one programming language or version to another, usually to make it compatible with older environments. For example, a transpiler can convert modern JavaScript (ES2025) syntax into the equivalent ES5 syntax so it runs in older browsers or older versions of Node.js.*
>
> > *A* **polyfill** *is a piece of code (typically a library or function) that adds functionality to older environments that don't natively support newer features. It mimics the behavior of new features, allowing developers to use them without breaking compatibility.*

All these challenges don't apply when developing applications with Node.js. Our Node.js applications usually run on a system and a Node.js runtime that are known in advance. This makes a big difference, as it allows us to write code for specific JavaScript and Node.js versions, ensuring there won't be any surprises when we run it on production machines.This, combined with the fact that newer versions of Node.js come with recent versions of V8, means we can confidently use most of the latest ECMAScript (ES) features without needing any extra transpilation steps. (ES is the standard on which the JavaScript language is based.)Bear in mind, however, that if we are developing a library meant for third-party use, we still need to consider that our code may run on different versions of Node.js. The common practice, in this case, is to target the oldest active **long-term support** (**LTS**) release and specify the `engines` section in our `package.json`. This way, the

package manager will warn users if they try to install a package that isn't compatible with their version of Node.js.

> *You can find out more about the Node.js release cycles at* `nodejsdp.link/node-releases`*. Also, you can find the reference for the* `engines` *section of* `package.json` *at* `nodejsdp.link/package-engines`*. Finally, you can get an idea of what ES feature is supported by each Node.js version at* `nodejsdp.link/node-green`*.*

The module system

From its inception, Node.js included a module system, even when JavaScript had no official support for one. The original Node.js module system is called CommonJS, and it uses the `require` keyword to import functions, variables, and classes from built-in modules or other modules on the device's filesystem.CommonJS was revolutionary for the JavaScript world, gaining popularity even on the client side, where it is used with module bundlers like Webpack or Rollup to create code bundles easily executable by the browser. CommonJS was essential for Node.js, allowing developers to create large, well-organized applications comparable to those on other server-side platforms.

> *A* **module bundler** *is a tool that combines multiple JavaScript files and their dependencies into a single or smaller number of files, often*

> *called bundles. This process helps optimize loading times and reduces the complexity of managing dependencies. Module bundlers analyze the relationships between various modules, resolve dependencies, and output them into a format that can be efficiently executed in a browser or other runtime environments.*

Today, JavaScript has the ES modules syntax (using the `import` keyword), which Node.js adopts in syntax only, as its underlying implementation differs from the browser's. While browsers mainly handle remote modules, Node.js, at least for now, deals only with modules on the local filesystem.We'll discuss modules in the context of Node.js in more detail in the next chapter.

Full access to operating system services

As we already mentioned, even if Node.js uses JavaScript, it doesn't run inside the boundaries of a browser. This allows Node.js to have bindings for all the major services offered by the underlying operating system.For example, we can access any file on the filesystem (subject to any operating system-level permission) thanks to the `fs` module, or we can write applications that use low-level TCP or UDP sockets thanks to the `net` and `dgram` modules. We can create HTTP(S) servers (with the `http` and `https` modules) or use the standard encryption and hashing algorithms of OpenSSL (with the `crypto` module). We can also access some of the V8 internals (the `v8` module) or run code in a different V8 context (with the

`vm` module).We can also run other processes (with the `child_process` module) or retrieve our own application's process information using the `process` global variable. In particular, from the `process` global variable, we can get a list of the environment variables assigned to the process (with `process.env`) or the command-line arguments passed to the application at the moment of its launch (with `process.argv`).Throughout the book, you'll have the opportunity to use many of the modules described here, but for a complete reference, you can check the official Node.js documentation at `nodejsdp.link/node-docs`.

Running native code

One of the most powerful capabilities offered by Node.js is certainly the possibility to create userland modules that can bind to native compiled code (e.g., written in compiled languages such as C, C++, or Rust). This gives the platform a tremendous advantage as it allows us to reuse existing or use new components written in performant compiled languages such as C/C++. Node.js officially provides great support for implementing native modules thanks to the Node-API interface.But what's the advantage? First of all, it allows us to reuse, with little effort, a vast amount of existing open-source libraries, and most importantly, it allows a company to reuse its own C/C++ legacy code without the need to migrate it.Another important consideration is that native code is still necessary to access low-level features such as

communicating with hardware drivers or with hardware ports (for example, USB or serial). In fact, thanks to its ability to link to native code, Node.js has become popular in the world of the **Internet of Things** (**IoT**) and homemade robotics.Finally, even though V8 is very (very) fast at executing JavaScript, it still has a performance penalty to pay compared to executing native code. In everyday computing, this is rarely an issue, but for CPU-intensive applications, such as those with a lot of data processing and manipulation, delegating the work to native code can make a lot of sense.We should also mention that, nowadays, most JavaScript **virtual machines** (**VMs**) (and also Node.js) support **WebAssembly** (**Wasm**), a low-level instruction format that allows us to compile languages other than JavaScript (such as C++ or Rust) into a format that is "executable" by JavaScript VMs. This brings many of the advantages we have mentioned, without the need to directly interface with native code.

> *You can learn more about Wasm on the official website of the project at* `nodejsdp.link/webassembly`.

Node.js and TypeScript

TypeScript is an open-source language developed by Microsoft to add a strong static type system to JavaScript. Think of TypeScript as an enhanced version of JavaScript with extra features. For instance, with TypeScript, developers can specify the types of arguments a function accepts, the type it returns, or the structure of an object. These types are checked before your

code runs, helping to catch issues like accessing non-existent properties or passing incorrect parameters. This makes your code more secure and robust, preventing many bugs before the code even goes live.It's important to know that TypeScript code can't run directly on JavaScript platforms like Node.js. Instead, TypeScript is used for static analysis and must be compiled (or "transpiled") into plain JavaScript to run. This process converts TypeScript into JavaScript files that can work in an environment supporting JavaScript. While this extra step might seem like more work, the advantages of catching errors early and enhancing code quality make it highly beneficial, especially for larger projects.

Using TypeScript with Node.js

If you want to use TypeScript with Node.js, you have a few options available.The first option is to use the official TypeScript compiler to convert your TypeScript files into equivalent JavaScript files that can be executed by Node.js. To install TypeScript in your project, run:

```
npm install --save-dev typescript
```

If you have a TypeScript file called `example.ts`, for example, you can transpile it to JavaScript with the following command:

```
npx tsc example.ts
```

This command will first check `example.ts` for any type error and, if everything is fine, it will convert it to plain JavaScript, creating a file called `example.js`. You can then execute `example.js` with Node.js:

```
node example.js
```

When developing an application, it can be tedious to manually transpile your code every time you want to run it. Fortunately, there are tools that handle this for you automatically, such as `ts-node` and `tsx`.To install `ts-node` or `tsx`, run:

```
npm install --save-dev ts-node
# or
npm install --save-dev tsx
```

Then you can directly execute `example.ts` with:

```
npx ts-node example.ts
# or
npx tsx example.ts
```

Additionally, `tsx` can also be used as a *Node.js loader*, which can be convenient in case you want to invoke the node CLI directly (e.g., if you need to pass additional CLI options arguments while executing your code):

```
node --import=tsx example.ts
```

> *Loaders are a mechanism that allows developers to customize how modules are loaded and processed in Node.js. By default, Node.js uses its built-in module loading system (CommonJS or ES modules), but with custom loaders, you can intercept and modify the original behavior. By using a custom TypeScript loader like* `tsx`*, you can intercept TypeScript modules during the loading process, transpile them into JavaScript, and then pass the resulting code to Node.js for execution. This eliminates the need for a separate build step, making development workflows smoother and more flexible.*
>
> > *Keep in mind that when using tools like* `ts-node` *and* `tsx` *to run TypeScript code directly, the code is being transpiled "on the fly." This means you incur the time cost of transpilation each time you run your code. While this approach is convenient for development, it's more efficient to pre-transpile your code before deploying it to production. This avoids the overhead of transpilation during runtime and ensures better performance.*

The Node.js core team is putting a lot of effort into making TypeScript a first-class citizen of the platform, so we can expect that it is going to become easier to execute TypeScript code in the future without having to install third-party tools.

The @types/node package

When developing TypeScript applications in a Node.js environment, you should use the `@types/node` package for a smooth development experience. This package provides TypeScript with the necessary type definitions for Node.js, enabling strong typing and enhanced autocompletion within your IDE. Node.js itself is written in JavaScript, which does not inherently include type definitions. Therefore, without `@types/node`, TypeScript would lack the knowledge of Node.js-specific globals, modules, and APIs, making it difficult to write type-safe code.The `@types/node` package includes type definitions for the entire Node.js API, covering everything from core modules like `fs`, `http`, and `path` to global objects like `process` and `Buffer`. By incorporating this package into your project, you gain access to TypeScript's powerful static type-checking features, which can help catch potential bugs early in the development process. Moreover, it provides comprehensive autocompletion in your code editor, making development faster and reducing the likelihood of errors caused by incorrect method usage or misconfigured parameters.To install the `@types/node package`, you can use npm:

```
npm install --save-dev @types/node
```

These commands add the `@types/node package` as a development dependency (i.e., it will be installed only during development but not in production environments).

TypeScript has a lot to offer, but it's not the main focus of this book. We'll provide TypeScript examples and tips when they help clarify specific topics, but for a comprehensive introduction to TypeScript, we recommend visiting the official TypeScript website at `nodejsdp.link/ts`*. If you're already familiar with TypeScript, you'll still find this book valuable. The patterns and techniques we discuss can be easily applied to any TypeScript project.*

Summary

In this chapter, you have seen how the Node.js platform is built upon a few important principles that shape both its internal architecture and the code we write. You have learned that Node.js has a minimal core and that embracing the "Node way" means writing modules that are smaller, simpler, and that expose only the minimum functionality necessary.Next, you discovered the reactor pattern, which is the pulsating heart of Node.js, and dissected the internal architecture of the platform runtime to reveal its other pillars: V8, libuv, bindings, and the core JavaScript library.Finally, we analyzed some of the main characteristics of using JavaScript in Node.js compared to the browser and learned how TypeScript can be leveraged when working with Node.js.Besides the obvious technical advantages enabled by its internal architecture, Node.js draws significant interest due to the principles it embodies and the vibrant community surrounding it. Its focus on simplicity and efficiency resonates with developers, offering a more human-

centered approach to programming that balances ease of use with scalability. This is why so many developers find themselves falling in love with Node.js.In the next chapter, we will go deep into one of the most fundamental and important topics of Node.js, its module system.

2

The Module System

In *Chapter 1*, *The Node.js Platform*, we briefly introduced the importance of modules in Node.js. We discussed how modules play a fundamental role in defining some of the pillars of the Node.js philosophy and its programming experience. However, what do we mean when we talk about modules and why are they so important?In generic terms, modules are the bricks for structuring non-trivial applications. Modules allow you to divide the code base into small units that can be developed and tested independently. Modules are also the main mechanism to enforce information hiding by keeping all the functions and variables that are not explicitly marked to be exported private. Additionally, modules make it easier to share and reuse code across projects.If you have experience with other programming languages, you have probably seen similar concepts being referred to with different names: **package** (Java, Go, PHP, Rust, or Dart), **assembly** (.NET), **gem** (Ruby), **crate** (Rust), or **unit** (Pascal dialects). The terminology is not perfectly interchangeable because every language or ecosystem comes with its own unique characteristics, but there is a significant overlap between these concepts.Due to historical reasons, Node.js has two different module systems: **ECMAScript modules** (**ES modules**) and **CommonJS**. Since ES modules are now the main module system in Node.js and are widely used on the web, we will focus on them in this chapter. We will briefly discuss CommonJS, as it is still found in older code bases and libraries, and it's useful to understand how to work with both module systems if needed. We'll also look at the reactor pattern and a few techniques for using and creating Node.js modules. By the end of this chapter, you should be able to make informed decisions on how to use and write modules effectively.Getting a good grasp of Node.js' module systems and module patterns is very important as we will rely on this knowledge in all the other chapters of this book.In short, these are the main topics we will be discussing throughout this chapter:

- Why modules are necessary
- The different module systems available in Node.js
- The revealing module pattern
- ES modules in Node.js
- CommonJS modules
- CommonJS and interoperability with ES modules
- How to use modules in TypeScript

Let's begin with why we need modules.

The need for modules

Let's start by clarifying the difference between *a module* and *a module system*. A module is the actual piece of software, while a module system is the syntax and tools that let us define and use modules in our projects.No matter the programming language or the platform, a good module system should help with some fundamental needs of software engineering:

- *Organizing code:* It should allow the code base to be split into multiple files. This keeps the code more organized and easier to understand, and enables independent development and testing of different parts.
- *Code reuse:* It should support code reuse across different projects. A module can provide a feature that's useful in various projects, making it simple to integrate this functionality wherever needed.
- *Encapsulation:* It should help hide implementation details and expose only a clear, simple interface. Most module systems allow certain parts of the code to remain *private* while providing *public* functions, classes, or objects for users of the module.
- *Dependency management:* It should make managing dependencies straightforward, enabling developers to build on existing modules (including third-party ones) and allowing users to easily import all necessary dependencies for a module.

Module systems in JavaScript and Node.js

Not all programming languages come with a built-in module system, and JavaScript lacked this feature for a long time after its inception.When writing JavaScript code for the browser, it is possible to split the code base into multiple files and then import them by using different `<script>` tags. For many years, this approach was good enough to build simple interactive websites, and JavaScript developers managed to get things done without having a fully-fledged module system.Only when JavaScript browser applications became more advanced and frameworks like *jQuery*, *Backbone*, and *AngularJS* took over the ecosystem did the JavaScript community come up with several initiatives aimed at defining a module system that could be effectively adopted within JavaScript projects. The most successful ones were **asynchronous module definition** (**AMD**), popularized by *RequireJS* (https://nodejsdp.link/requirejs), and later **universal module definition** (**UMD**—https://nodejsdp.link/umd).When Node.js was created, it was conceived as a server runtime for JavaScript with direct access to the underlying filesystem so there was a unique opportunity to introduce a different way to manage modules. The idea was not to rely on HTML `<script>` tags and resources accessible through URLs. Instead, the choice was to rely purely on JavaScript files available on the local filesystem. For its module system, Node.js came up with an implementation of the *CommonJS* specification (https://nodejsdp.link/commonjs), which was designed to provide a module system for JavaScript in *browserless* environments. It's worth noting that CommonJS was not part of the official ECMAScript standard, but an independent initiative to standardize JavaScript outside the browser.CommonJS has been the dominant module system in Node.js for many years. It became so popular that people started using it to write modules that could be used even in browser applications thanks to *module bundlers* like *Browserify* (https://nodejsdp.link/browserify) and *Webpack* (https://nodejsdp.link/webpack).In 2015, with the release of

ECMAScript 2015 (*ES2015*), there was an official proposal to define a standard module system for JavaScript: *ES modules*. ES modules bring a lot of innovation in the JavaScript ecosystem, and, among other things, they try to bridge the gap between how modules are managed on browsers and servers.ES2015 defined only the formal specification for ES modules in terms of syntax and semantics, but it didn't provide any implementation details. It took several years for browser companies and the Node.js community to develop strong implementations, with Node.js offering stable support for ES modules starting from version 13.2 (released in 2019). As a result, the transition from CommonJS to ES modules has been somewhat slow and, during the transition years, developers have been using various techniques to publish *dual-mode* libraries that can work with both ES modules and CommonJS.Today, ES modules are the widely accepted standard module system for JavaScript and Node.js. While CommonJS is still common in legacy code bases and older libraries, most new projects are now written using ESM, and CommonJS usage is expected to decline over time. This book uses ES modules for all code examples, but we will also examine some CommonJS code to understand how the two module systems can work together.

The revealing module pattern

Before diving straight into ES modules, it’s worth taking a brief detour to explore a foundational JavaScript pattern, the **revealing module pattern**, a pattern that facilitates information hiding and will be useful in building a primitive module system. This background will not only deepen our appreciation of fully-fledged module systems like ES modules and CommonJS but also provide a great opportunity to see how these module systems can be implemented behind the scenes. As such, this pattern will become a foundational piece of knowledge that will help us to deeply understand ES modules.As we said, modules are the bricks for structuring non-trivial applications and the main mechanism to enforce information hiding by keeping all the functions and variables that are not explicitly marked to be exported private.One major issue with JavaScript in the browser is the lack of namespacing. Since every script runs in the global scope, both internal application code and third-party dependencies can clutter the scope by exposing their own functions and variables. This can be very problematic. For example, if a third-party library creates a global variable called `utils` and another library or the application code itself accidentally overwrites or changes `utils`, the code that depends on it might crash in unpredictable ways. Similar issues can arise if other libraries or the application code accidentally call a function that was intended only for internal use by another library.Relying on the global scope is risky, especially as your application grows and you depend more on code written by others.A popular technique to solve this class of problems is called the *revealing module pattern*, and it looks like this:

```
const myModule = (() => {
  const privateFoo = () => {}
  const privateBar = []
  console.log('Inside:', privateFoo, privateBar)
  const exported = {
    publicFoo: () => {},
```

```
    publicBar: () => {},
  }
  return exported
})() // once the parenthesis here are parsed
    // the function will be invoked
    // and the returned value assigned to myModule
console.log('Outside:', myModule.privateFoo, myModule.privateBar)
console.log('Module:', myModule)
```

This pattern leverages a self-invoking function. This type of function is sometimes also referred to as an **immediately invoked function expression** (**IIFE**) and it is used to create a private scope, exporting only the parts that are meant to be public.In JavaScript, variables created inside a function are not accessible from the outer scope (outside the function). Functions can use the `return` statement to selectively propagate information to the outer scope.This pattern essentially exploits these properties to keep the private information hidden and export only a public-facing API.In the preceding code, the `myModule` variable contains only the exported API, while the rest of the module content is practically inaccessible from the outside.The `log` statement is going to print something like this:

```
Inside: [Function: privateFoo] []
Outside: undefined undefined
Module: { publicFoo: [Function: publicFoo], publicBar: [Function: publicBa
```

This demonstrates a few important details:

- The private `privateFoo` and `privateBar` variables are accessible from within the immediately invoked function.
- Printing `myModule` doesn't show `privateFoo` and `privateBar` among the members of the object and we have no way to access these values from outside the immediately invoked function. If we try to access `myModule.privateFoo` and `myModule.privateBar`, these values are `undefined`.
- Only the `exported` properties `publicFoo` and `publicBar` are directly accessible from `myModule`.

The revealing module pattern can be used to create a basic module system, and this concept is actually the foundation of the CommonJS module system. In this chapter, the revealing module pattern helps demonstrate how certain intrinsic features of JavaScript can be used to structure code modularly and implement information hiding. However, in practice, you're not expected to build your own module system from scratch; instead, you'll be using ES modules.

ES modules

ES modules were introduced as part of the ES2015 specification with the goal of giving JavaScript an official module system suitable for different execution environments. The ES modules specification tries to retain some good ideas from previous existing module systems like CommonJS and AMD. The syntax is very simple and compact. There is support for cyclic dependencies and the possibility to load modules asynchronously.

> *A cyclic dependency in a module system occurs when two or more modules depend on each other in a circular manner. For example, Module A imports from Module B, and Module B, in turn, imports from Module A. This creates a loop in dependencies that can lead to issues like incomplete module loading, runtime errors, or unexpected behavior, as the modules are unable to fully resolve due to their mutual reliance.*

Using ES modules in Node.js

As previously discussed, when ES modules were introduced in Node.js, CommonJS had already long been the default module system. As a result, support for ES modules in Node.js had to be added carefully to maintain backward compatibility. This means that ES modules are not the default and must be explicitly enabled. To adopt ES modules in a project, there are specific steps that signal to Node.js that a file or module should use the ES module syntax, making it an opt-in feature for developers.Node.js will consider every `.js` file to be written using the CommonJS syntax by default. For example, let's say that we have a file called `index.js` with the following content:

```
import { someFeature } from './someModule.js'
console.log(someFeature)
```

We could then try to execute it with the following:

```
node index.js
```

You might see an error that looks like the following:

```
(node:69441) Warning: To load an ES module, set "type": "module" in the pac
(Use `node --trace-warnings ...` to show where the warning was created)
index.js:1
import { someFeature } from './someModule.js'
^^^^^^
SyntaxError: Cannot use import statement outside a module
```

This message means that our `index.js` file isn't recognized as an ES module, so we can't use ES module syntax. Node.js also provides helpful suggestions on how to fix this: we need to tell Node.js to load this file as an ES

module. There are a few ways we can do that:

- Give the module file the extension `.mjs` (note that, alternatively, you can use `.cjs` to force the module to be interpreted as a CommonJS module).
- Add a field called `"type"` with a value of `"module"` to the nearest parent `package.json` as in the following example:

```
{
  "name": "sample-esm-project",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module"
}
```

- Use the flag `--experimental-default-type="module"`. This flag sets the default module system to ES modules, making it equivalent to adding `"type": "module"` in your `package.json`. However, you'll need to specify this flag explicitly each time you run `node` to execute an ES module.
- Use the flag `--experimental-detect-module`. This flag instructs Node.js to analyze ambiguous files (such as those with a `.js` extension) when the module system is not explicitly specified. Node.js will attempt to infer the module type by scanning the file for keywords like `import` and `export`, which typically indicate an ES module rather than CommonJS.

  Our current recommendation is to use `"type":"module"` in the `package.json`, so you can keep using the `.js` extension which is the most commonly supported across text editors. Chances are that in the future Node.js will default to ES modules or that it will automatically detect the correct module system to use to load your files (at the time of writing, Node.js 23 has adopted module detection by default, so this is likely to become the default behavior in the future). Throughout this book, we will be using our recommended approach for most of our examples, so if you are copying and pasting examples straight from the book, make sure that you also create a `package.json` file with the `"type":"module"` entry.

Let's now have a look at the ES module syntax.

The ES module syntax

In this section, we'll focus specifically on the syntax of ES modules, covering its core elements like named and default exports, mixed exports, and module identifiers. We'll also look at how ES module syntax supports both static imports (loaded at the start of execution) and dynamic imports, which can be loaded conditionally or asynchronously.

Named exports and imports

ES modules allow us to export constants, functions, and classes from a module through the `export` keyword.In an ES module, everything is private by default and only exported entities are publicly accessible from other modules.The `export` keyword can be used in front of the entities that we want to make available to the module users. Let's see an example:

```
// logger.js
// exports a function as `log`
export function log(message) {
  console.log(message)
}
// exports a constant as `DEFAULT_LEVEL`
export const DEFAULT_LEVEL = 'info'
// exports an object as `LEVELS`
export const LEVELS = {
  error: 0,
  debug: 1,
  warn: 2,
  data: 3,
  info: 4,
  verbose: 5
}
// exports a class as `Logger`
export class Logger {
  constructor(name) {
    this.name = name
  }
  log(message) {
    console.log(`[${this.name}] ${message}`)
  }
}
```

If we want to import entities from a module, we can use the `import` keyword. The syntax is quite flexible, and it allows us to import one or more entities and even rename imports. Let's see some examples:

```
// main.js
import * as loggerModule from './logger.js'
console.log(loggerModule)
```

In this example, we are using the `*` syntax (also called **namespace import**) to import all the members of the module and assign them to the local `loggerModule` variable. This example will output something like this:

```
[Module] {
  DEFAULT_LEVEL: 'info',
  LEVELS: { error: 0, debug: 1, warn: 2, data: 3, info: 4,
    verbose: 5 },
  Logger: [Function: Logger],
  log: [Function: log]
}
```

As we can see, all the entities exported in our module are now accessible in the `loggerModule` namespace. For instance, we could refer to the `log()` function through `loggerModule.log`.It's generally better to be explicit and to avoid importing an entire module and instead import only the specific entities that are needed in the current context:

```
import { log } from './logger.js'
log('Hello World')
```

If we want to import more than one entity, this is how we would do that:

```
import { Logger, log } from './logger.js'
log('Hello World')
const logger = new Logger('DEFAULT')
logger.log('Hello world')
```

When we use this type of `import` statement, the entities are imported into the current scope, so there is a risk of a name clash. The following code, for example, would not work:

```
import { log } from './logger.js'
const log = console.log
```

If we try to execute the preceding snippet, the interpreter fails with the following error:

```
SyntaxError: Identifier 'log' has already been declared
```

In situations like this one, we can resolve the clash by renaming the imported entity with the `as` keyword:

```
import { log as log2 } from './logger.js'
const log = console.log
log('message from log')
log2('message from log2')
```

This approach can be particularly useful when the clash is generated by importing two entities with the same name from different modules, and therefore, changing the original names is outside the consumer's control. It can also be useful when you want to use a shorter or cleaner name for the entities you are importing.

> *Note that if you try to import a module member that is not exported, this will result in a syntax error at run-time. For example, we might try to execute the following code:*
>
> `Import { something } from './logger.js'`
>
> *We will then get the following error:*
>
> `SyntaxError: The requested module './logger.js' does not provide an expor`

Default exports and imports

Sometimes, you might want to export a single unnamed entity. This can be convenient as it encourages module developers to follow the single-responsibility principle and expose only one clear interface. With ES modules, we can do that with a **default export**. A default export makes use of the `export default` keywords and it looks like this:

```
// logger.js
export default class Logger {
  constructor(name) {
    this.name = name
  }
  log(message) {
    console.log(`[${this.name}] ${message}`)
  }
}
```

In this case, the name `Logger` is ignored, and the entity exported is registered under the name `default`. This exported name can be imported as follows:

```
// main.js
import MyLogger from './logger.js'
```

```
const logger = new MyLogger('info')
logger.log('Hello World')
```

The difference with named ES module imports is that here, since the default export is considered unnamed, we can import it and at the same time assign it a local name of our choice. In this example, we can replace `MyLogger` with anything else that makes sense in our context. Note also that we don't have to wrap the import name around brackets or use the `as` keyword when renaming.Internally, a default export is equivalent to a named export with `default` as the name. We can easily verify this statement by running the following snippet of code:

```
// showDefault.js
import * as loggerModule from './logger.js'
console.log(loggerModule.default)
```

When executed, the previous code will print something like this:

```
 [class Logger]
```

One thing that we cannot do, though, is import the default entity explicitly. In fact, something like the following will fail:

```
import { default } from './logger.js'
```

The execution will fail with a `SyntaxError: Unexpected reserved word` error. This happens because the `default` keyword cannot be used as a variable name. It is valid as an object attribute, so in the previous example, it is okay to use `loggerModule.default`, but we can't have a variable named `default` directly in the scope.

Mixed exports

It is possible to mix named exports and a default export within an ES module. Let's have a look at an example:

```
// logger.js
export default function log(message) {
  console.log(message)
}
export function info(message) {
  log(`info: ${message}`)
}
```

The preceding code is exporting the `log()` function as a default export and a named export for a function called `info()`.

> *Note that `info()` can reference `log()` internally. It would not be possible to replace the call to `log()` with `default()` to do that, as it would be a syntax error (unexpected token `default`).*

If we want to import both the default export and one or more named exports, we can do it using the following format:

```
import mylog, { info } from './logger.js'
```

In the previous example, we are importing the default export from `logger.js` (as `mylog`) and the named export `info`.

> *Note that, with this syntax, the default member should always come first. Writing the following code would result in a syntax error:*
>
> ```
> Import { info }, mylog from './logger.js'
> ```

Let's now discuss some key details and differences between the default export and named exports:

- The default export is a convenient mechanism to communicate what is the single most important functionality for a module. Also, from the perspective of the user, it can be easier to import the obvious piece of functionality without having to know the exact name of the binding.
- Named exports are explicit. Having predetermined names allows IDEs to support the developer with automatic imports, autocomplete, and refactoring tools. For instance, if we type `writeFileSync`, the editor might automatically add `import { writeFileSync } from 'node:fs'` at the beginning of the current file. Default exports, on the contrary, make all these things more complicated as a given functionality could have different names in different files, so it's harder to make inferences on which module might provide a given functionality based only on a given name.
- In some circumstances, default exports might make it harder to apply dead code elimination (tree shaking). For example, a module could provide only a default export, which is an object where all the functionality is exposed as properties of such an object. When we import this default object, most module bundlers will consider the entire object being used and they won't be able to eliminate any unused code from the exported functionality.

The drawbacks of using default exports in JavaScript are greater than the benefits, leading the community to generally favor avoiding them. Some code linters now include rules to detect and warn against the use of default exports.

> *Biome (https://nodejsdp.link/biome), a JavaScript linter and formatter, enables a linting rule to disable default exports by default. The documentation page for this rule provides some additional details on why this is to be considered good practice:*
>
> *https://nodejsdp.link/no-default-export*

This is not a hard rule and there are notable exceptions to this suggestion. For instance, all Node.js core modules have both a default export and a number of named exports. Also, React (https://nodejsdp.link/react) uses mixed exports.Consider carefully what the best approach for your specific module is and what you want the developer experience to be for the users of your module.

Module identifiers

Module identifiers (also called *module specifiers*) are the different types of values that we can use in our `import` statements to specify the location of the module we want to load.So far, we have only seen relative paths, but there are several other possibilities and some nuances to keep in mind. Let's list all the possibilities:

- *Relative specifiers* like `./logger.js` or `../logger.js` are used to refer to a path relative to the location of the importing file.
- *Absolute specifiers* like `file:///opt/nodejs/config.js` or `/opt/nodejs/config.js` refer directly and explicitly to a full path.
- *Bare specifiers* are identifiers like `fastify` or `http`. They represent modules available in the `node_modules` folder and generally installed through a package manager (such as npm) or available as core Node.js modules.

  > *To avoid ambiguity between core Node.js modules and third-party modules, Node.js supports an optional `node:` prefix (e.g., `node:http`). This is the recommended way to import Node.js core modules.*

- *Deep import specifiers* like `fastify/lib/logger.js` refer to a path within a package in `node_modules` (`fastify`, in this case).

In browser environments, it is possible to import modules directly by specifying the module URL, for instance, https://unpkg.com/lodash. This feature is not supported by Node.js.

Static and dynamic imports

ES modules are *static*, which means that imports are described at the top level of every module and outside any control flow statement. Also, the name of the imported modules cannot be dynamically generated at runtime using expressions; only constant strings are allowed.For instance, the following code wouldn't be valid when using ES modules:

```
if (condition) {
  import module1 from 'module1'
} else {
  import module2 from 'module2'
}
```

At first glance, these characteristics of ES modules might seem like unnecessary limitations, but having static imports opens up a number of interesting scenarios. For instance, static imports allow the static analysis of the dependency tree, which allows optimizations such as dead code elimination (tree shaking).Does this mean that with ES modules, we can't create module identifiers at runtime or import modules conditionally? For example, what if we want to load a specific module—perhaps a heavy one—only when the user accesses the feature that needs it? Or what if we need to import a particular translation module based on the user's language, or a version of a module that depends on the user's operating system?Fortunately, ES modules include a feature called *dynamic imports* (or *async imports*) that addresses these scenarios.Async imports can be performed at runtime using the special `import()` operator.The `import()` operator is syntactically equivalent to a function that takes a module identifier as an argument. It returns a promise that resolves to a module object.

> *We will learn more about promises in Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await, so don't worry too much about understanding all the nuances of the specific promise syntax for now.*

The module identifier can be any module identifier supported by static imports as discussed in the previous section. Now, let's see how to use dynamic imports with a simple example.We want to build a command line application that can print "*Hello World*" in different languages. In the future, we will probably want to support many more phrases and languages, so it makes sense to have one file with the translations of all the user-facing strings for each supported language.Let's create some example modules for some of the languages we want to support:

```
// strings-el.js
export const HELLO = 'Γεια σου κόσμε'
// strings-en.js
export const HELLO = 'Hello World'
// strings-es.js
export const HELLO = 'Hola mundo'
// strings-it.js
export const HELLO = 'Ciao mondo'
// strings-pl.js
export const HELLO = 'Witaj świecie'
```

Now let's create the main script that takes a language code from the command line and prints "Hello World" in the selected language:

```
// main.js
const SUPPORTED_LANGUAGES = ['el', 'en', 'es', 'it', 'pl'] // (1)
const selectedLanguage = process.argv[2] // (2)
if (!selectedLanguage) { // (3)
  console.error(
    `Please specify a language

    Usage: node ${process.argv[1]} <language_code>
    Supported languages: ${SUPPORTED_LANGUAGES.join(', ')}`
  )
  process.exit(1)
}
if (!SUPPORTED_LANGUAGES.includes(selectedLanguage)) { // (4)
  console.error('The specified language is not supported')
  process.exit(1)
}
const translationModule = `./strings-${selectedLanguage}.js` // (5)
const strings = await import(translationModule) // (6)
console.log(strings.HELLO) // (7)
```

The first part of the script is quite simple. What we do there is as follows:

1. Define a list of supported languages.
2. Read the selected language from the first argument passed in the command line.
3. Validate that the user has passed the argument and if not, provide a helpful error message.
4. Finally, we handle the case where the selected language is not supported.

The second part of the code is where we actually use dynamic imports:

1. First of all, we dynamically build the name of the module we want to import based on the selected language. Note that the module name needs to be a relative path to the current module file; that's why we are prepending `./` to the filename.
2. We use the `import()` operator to trigger the dynamic import of the module. This import happens asynchronously, returning a promise. We use `await` to wait for the promise to resolve and store the resolved value in the `strings` variable.
3. Now, we can access `strings.HELLO` and print its value to the console.

We can execute this script like this:

```
node main.js it
```

We should see *Ciao mondo* being printed to our console.

The module resolution algorithm

The term *dependency hell* describes a scenario where two or more dependencies in a program rely on a shared library but require different, incompatible versions of it. This is a common problem in many programming languages and is often difficult to resolve.Node.js addresses this problem elegantly by loading different versions of a module based on where the module is loaded from. This capability is largely due to how Node.js package managers (such as npm or pnpm) organize an application's dependencies and the resolving algorithm that maps a module specifier to a file in the file system. For example, this algorithm translates an identifier such as `'fastify/lib/logger.js'` to a URL that represents a file that can be loaded from the filesystem such as `file:///Users/luciano/projects/example_web_server/node_modules/fastify/lib/lo`, allowing the file to be loaded correctly.Node.js implements the ES module function `import.meta.resolve()`, which allows us to see how a given module specifier is resolved in the current context. Let's try to have a high-level overview of how the module resolution algorithm works. We can split the algorithm into the following three major branches:

- **File modules**: If the module specifier starts with `/`, it is considered an absolute path to the module file in the filesystem. The path is normalized and converted into a URL with a `file://` prefix. If it starts with `./` or `../`, then the module specifier is considered a relative path, which is calculated starting from the directory of the requiring module.
- **Node.js core modules**: If the module specifier is not prefixed with `/`, `./`, or `../`, the algorithm will first try to search within the core Node.js modules. If the module specifier matches one of the Node.js core modules, then the given specifier is prefixed with `node:` and returned. If the provided module specifier is already prefixed with `node:` then it is returned as is.
- **Package modules**: If no core module is found matching the given module specifier, then the search continues by looking for a matching module in the first `node_modules` directory that is found navigating up in the directory structure starting from the importing module. The algorithm continues to search for a match by looking into the next `node_modules` directory up in the directory tree until it reaches the root of the filesystem.

  There are some additional types of specifiers supported such as the `data:` prefix. The complete, formal documentation of the resolving algorithm can be found at https://nodejsdp.link/resolve_esm.

We can see this algorithm in action with the following code:

```
// relative import
console.log(import.meta.resolve('./utils/example.js'))
  // -> file://<project_path>/utils/example.js
// Node.js core module import
console.log(import.meta.resolve('assert'))
  // -> node:assert
console.log(import.meta.resolve('node:assert'))
  // -> node:assert
// Third-party library import (with fastify@5.1.0 installed)
console.log(import.meta.resolve('fastify/lib/logger.js'))
  // -> file://<project_path>/node_modules/fastify/lib/logger.js
```

The `node_modules` directory is actually where the package managers install the dependencies of each package. This means that, based on the algorithm we just described, each package can have its own private dependencies. For example, consider the following directory structure:

```
myApp
├── foo.js
└── node_modules
    ├── depA
    │   └── index.js
    ├── depB
    │   ├── bar.js
    │   └── node_modules
    │       └── depA
    │           └── index.js
    └── depC
        ├── foobar.js
        └── node_modules
            └── depA
                └── index.js
```

In the previous example, `myApp`, `depB`, and `depC` all depend on `depA`. However, they all have their own private version of the dependency! Following the rules of the resolving algorithm, importing `depA` will load a different file depending on the module that requires it. Let's look at some examples:

- Importing `depA` from `/myApp/foo.js` will load `/myApp/node_modules/depA/index.js.`
- Importing `'depA'` from `/myApp/node_modules/depB/bar.js` will load `/myApp/node_modules/depB/node_modules/depA/index.js.`

- Importing `depA` from `/myApp/node_modules/depC/foobar.js` will load `/myApp/node_modules/depC/node_modules/depA/index.js.`

The resolving algorithm is the core part behind the robustness of the Node.js dependency management, and it makes it possible to have hundreds or even thousands of packages in an application without having collisions or problems of version compatibility between the installed packages.

Module loading in depth

To understand how ES modules work and how they can deal effectively with dependencies and even circular dependencies, we must deep dive a little bit deeper into how JavaScript code is parsed and evaluated when using ES modules.In this section, we will learn how ES modules are loaded, present the idea of **read-only live bindings**, and finally, discuss an example with circular dependencies.

Loading phases

Node.js uses dependency graphs to load modules in the correct order. The interpreter builds this graph comprising all the necessary modules. By respecting dependencies, this graph prevents errors and guarantees that each module is available when needed.

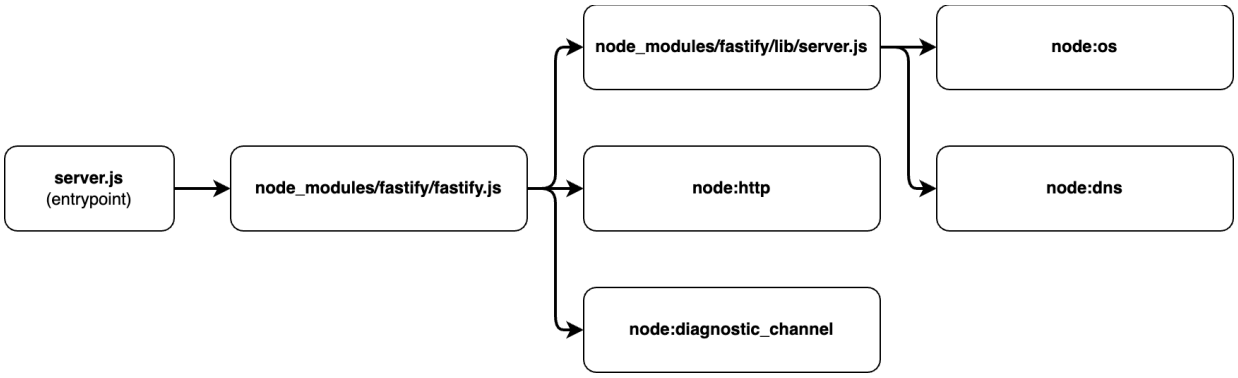

Figure 2.1: An example of Depdency Graph for a web server application using Fastify

In *Figure 2.1*, we see an example of a dependency graph. Each node in the graph represents a module, and arrows indicate when one module depends on another. The main module, `server.js` (also called the **entry point**), depends on `fastify`. Fastify itself relies on several modules: `node:http`, `node:diagnostic_channel`, and an internal module (`fastify/lib/server.js`). This internal module, in turn, depends on `node:os` and `node:dns`. Note that this representation is only a partial picture for illustrative purposes; in a real Fastify-based project, there will likely be many more dependencies.

> *In generic terms, a* **dependency graph** *can be defined as a* **directed graph** *(https://nodejsdp.link/directed-graph) representing the dependencies of a group of objects. In the context of this section, when we refer to a dependency graph, we want to indicate the dependency relationship between ES modules. As we will see, using a dependency graph allows us to determine the order in which all the necessary modules should be loaded in a given project.*

Essentially, the dependency graph is needed by the interpreter to figure out how modules depend on each other and in what order the code needs to be executed. When the `node` interpreter is launched, it gets passed some code to execute, generally in the form of a JavaScript file. This file is the starting point for the dependency resolution, and it is called the **entry point**. From the entry point, the interpreter will find and follow all the `import` statements recursively in a depth-first fashion, until all the necessary code is explored and then evaluated.This process happens in three separate phases:

- **Phase 1—construction (or parsing)**: The interpreter identifies all imports and recursively loads the content of each module from their respective files.
- **Phase 2—instantiation**: For each exported entity in every module, the interpreter creates a named reference in memory, but it does not assign it a value yet. References are created for all the `import` and `export` statements to track the dependency relationships between them (**linking**). No JavaScript code is executed during this phase.
- **Phase 3—evaluation**: The Node.js interpreter executes the code so that all the previously instantiated entities can get an actual value. Now, running the code starting from the entry point is possible because all the blanks have been filled.

We could say that Phase 1 is about finding all the dots, Phase 2 connects those dots creating paths, and finally, Phase 3 walks through the paths in the right order.Since these three phases are separate, no code can be executed until the entire dependency graph is fully constructed. As a result, module imports and exports must be static. We'll dive deeper into this process when we discuss how ES modules manage circular dependencies, so don't worry if the details aren't fully clear yet.

> *This process differs significantly from how CommonJS manages dependencies. CommonJS is dynamic; when a module is required, its content is immediately loaded and executed. This flexibility allows `require` to be used within `if` statements or loops, and module identifiers can be created from variables. However, this dynamic approach makes it challenging for CommonJS to handle cyclic dependencies effectively — a challenge that, as we'll see, ES modules manage more efficiently.*

Read-only live bindings

Another fundamental characteristic of ES modules, which helps with cyclic dependencies, is the idea that imported modules are effectively *read-only live bindings* to their exported values.Let's clarify what this means with a simple example:

```
// counter.js
export let count = 0
export function increment() {
  count++
}
```

This module exports two values: a simple integer counter called `count` and an `increment` function that increases the counter by one.Let's now write some code that uses this module:

```
// main.js
import { count, increment } from './counter.js'
console.log(count) // prints 0
increment()
console.log(count) // prints 1
count++ // TypeError: Assignment to constant variable!
```

What we can see in this code is that we can read the value of `count` at any time and change it using the `increment()` function, but as soon as we try to mutate the `count` variable directly, we get an error as if we were trying to mutate a `const` binding.This proves that when an entity is imported in the scope, the binding to its original value cannot be changed (*read-only binding*) unless the bound value changes within the scope of the original module itself (*live binding*), which is outside the direct control of the consumer code.

> *This approach is fundamentally different from CommonJS. In fact, in CommonJS, the entire* `exports` *object is copied (shallow copy) when required from a module. This means that, if the value of primitive variables like numbers or strings are changed later, the requiring module won't be able to see those changes.*

Circular dependencies

Many consider circular dependencies an intrinsic design issue, but it is something that might actually happen in a real project, so it's useful for us to know how this works.Let's walk through an example together to see how ES modules behave when dealing with circular dependencies. Let's suppose we have the scenario represented in *Figure 2.2*:

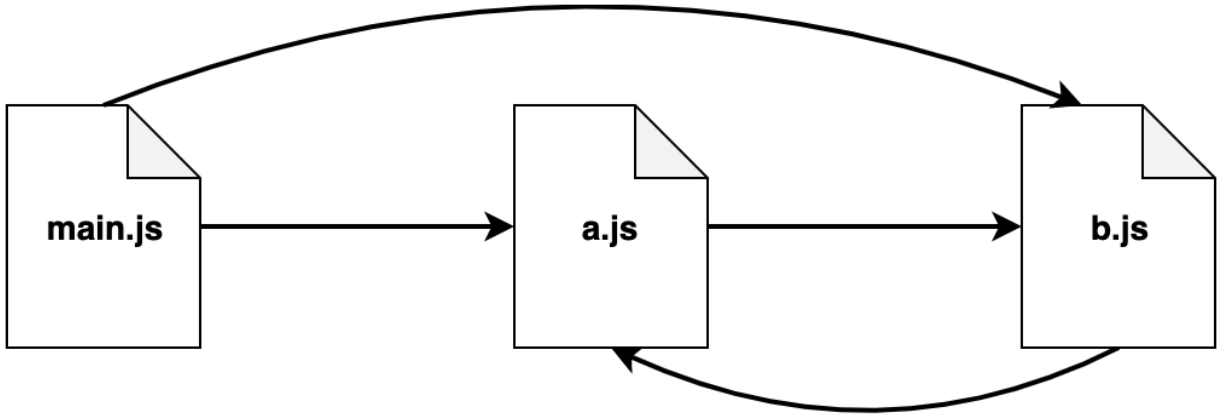

Figure 2.2: An example of circular dependency

A module called `main.js` imports `a.js` and `b.js`. In turn, `a.js` imports `b.js`. However, `b.js` relies on `a.js` as well! It's obvious that we have a circular dependency here as module `a.js` requires module `b.js` and module `b.js` requires module `a.js`.Let's implement modules `a.js` and `b.js` first:

```
// a.js
import * as bModule from './b.js'
export let loaded = false
export const b = bModule
loaded = true
// b.js
import * as aModule from './a.js'
export let loaded = false
export const a = aModule
loaded = true
```

Note that both `a.js` and `b.js` export a binding called `loaded`, which we can use to observe what happens during the loading of each module. The value is initially set to `false`, and when the module is executed, it is immediately updated to `true`. It's also important to note that `a.js` imports `b.js`, and `b.js` imports `a.js` in return, creating the circular dependency.Now let's see how to import those two modules in our `main.js` file (the entry point):

```
// main.js
import * as a from './a.js'
import * as b from './b.js'
console.log('a ->', a)
console.log('b ->', b)
```

When we run `main.js`, we will see the following output:

```
a -> <ref *1> [Module: null prototype] {
  b: [Module: null prototype] { a: [Circular *1], loaded: true },
  loaded: true
}
b -> <ref *1> [Module: null prototype] {
  a: [Module: null prototype] { b: [Circular *1], loaded: true },
  loaded: true
}
```

The interesting part here is that `a.js` and `b.js` have a complete view of each other. We can observe that `loaded` is set to `true` in all references to both `a` and `b`. This makes even more sense when we realize that `b` within `a`, and `a` within `b`, are actual references to the respective modules, not copies. There's no data duplication during the import and execution of the modules. If we were to swap the order of the imports in `main.js`, the output would remain the same.

> *If we implemented this same example using CommonJS, the result would be significantly different: the inner `loaded` values would be `false`. Additionally, if we swapped the order of imports in CommonJS, we would see a different output. These differences arise from the dynamic nature of CommonJS and the shallow copying that occurs when entities are imported.*

To clearly understand how ES modules make it possible to support circular dependencies, it's worth spending some more time observing what happens in the three phases of the module resolution (parsing, instantiation, and evaluation) for this specific example.

Phase 1—parsing

During the parsing phase, the code is explored starting from the entry point (`main.js`). The interpreter looks only for `import` statements to find all the necessary modules and to load the source code from the module files. The dependency graph is explored in a depth-first fashion, and every module is visited only once. This way, the interpreter builds a view of the dependencies in a way that looks like a tree structure, as shown in *Figure 2.3*:

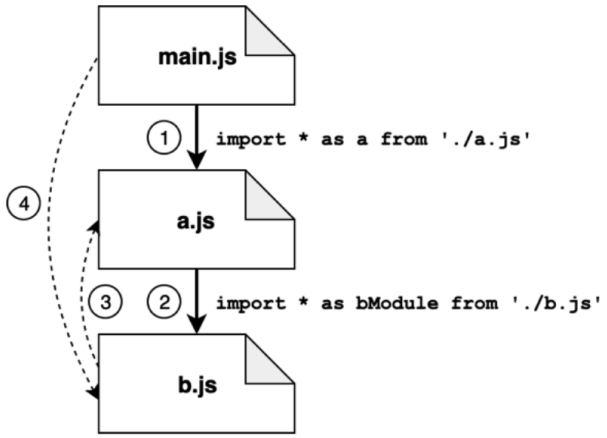

Figure 2.3: Parsing of cyclic dependencies with ES modules

Given the example in *Figure 2.3*, let's discuss the various steps of the parsing phase:

1. From `main.js`, the first import found leads us straight into `a.js`.
2. In `a.js`, we find an import pointing to `b.js`.
3. In `b.js`, we also have an import back to `a.js` (our cycle), but since `a.js` has already been visited, this path is not explored again.
4. At this point, the exploration starts to wind back: `b.js` doesn't have other imports, so we go back to `a.js`; `a.js` doesn't have other `import` statements, so we go back to `main.js`. Here, we find another import pointing to `b.js`, but again, this module has been explored already, so this path is ignored.

   To clarify the depth-first approach, imagine `main.js` also imports a module called `c.js`, listed after `a.js`. With this setup, `c.js` would only be parsed once both `a.js` and `b.js` are fully processed, and the parsing process returns to `main.js` to handle the remaining imports. The general idea behind a depth-first traversal is to explore each branch as deeply as possible before backtracking to the parent node to explore other paths. If this concept is still unclear, consider reviewing this link on **depth-first search (DFS)** *for more examples and illustrations:*

   https://nodejsdp.link/dfs

At this point, our depth-first visit of the dependency graph has been completed and we have a linear view of the modules, as shown in *Figure 2.4*:

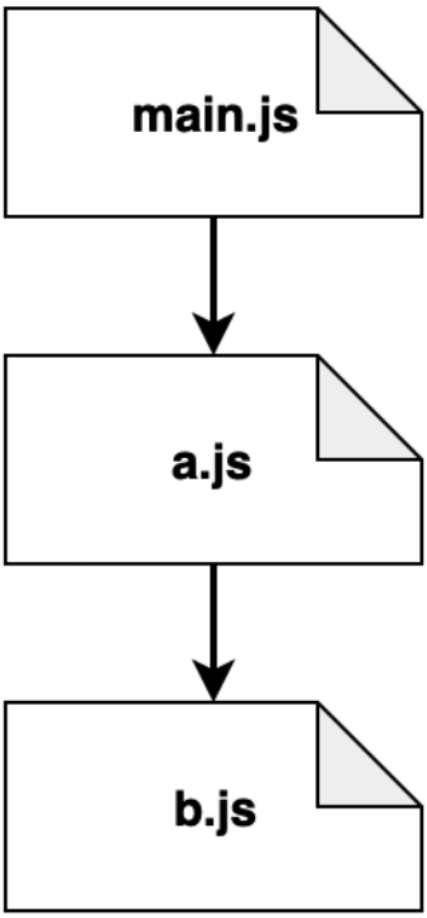

Figure 2.4: A linear view of the module graph where cycles have been removed

This particular view is quite simple. In more realistic scenarios with a lot more modules, the view will look more like a tree structure.

Phase 2—instantiation

In the instantiation phase, the interpreter walks the tree view obtained from the previous phase from the bottom to the top. For every module, the interpreter will look for all the exported properties first and build out a map of the exported names in memory:

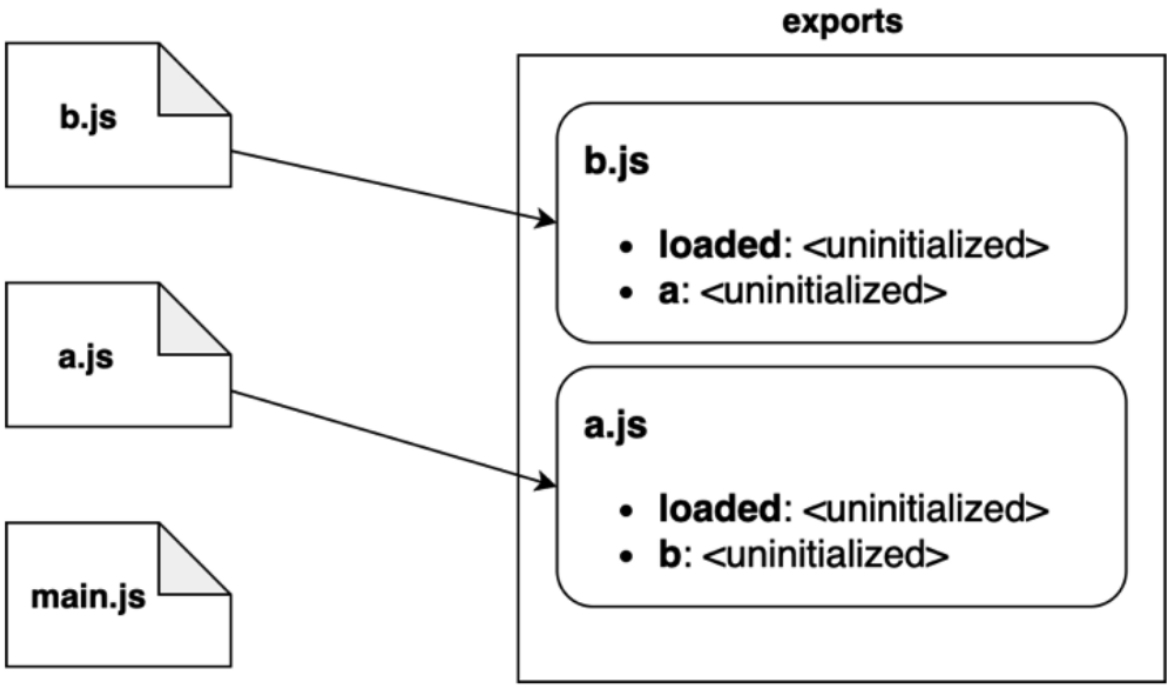

Figure 2.5: A visual representation of the instantiation phase

Figure 2.5 describes the order in which every module is instantiated:

1. The interpreter starts from `b.js` and discovers that the module exports `loaded` and `a`.
2. Then, the interpreter moves to `a.js`, which exports `loaded` and `b`.
3. Finally, it moves to `main.js`, which does not export any functionality.

> *Note that, in this phase, the exports map only keeps track of the exported names; their associated values are considered uninitialized for now.*

After this sequence of steps, the interpreter will do another pass to link the exported names to the modules importing them, as shown in *Figure 2.6*:

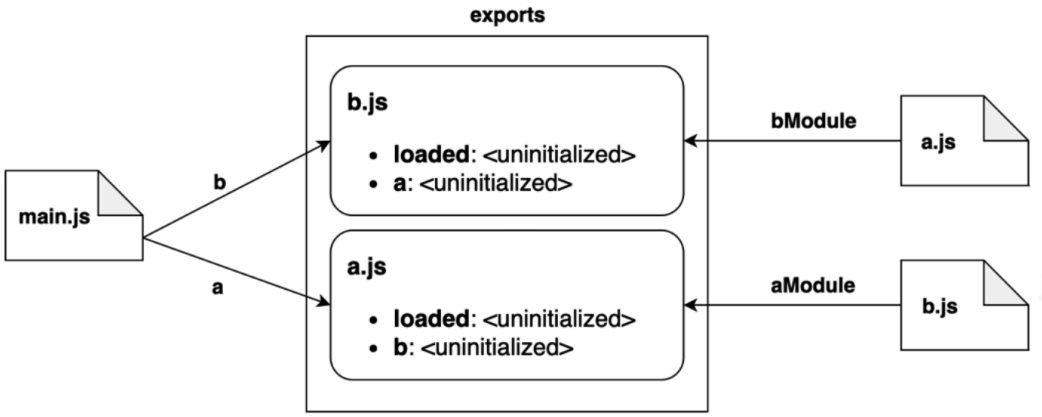

Figure 2.6: Linking exports with imports across modules

We can describe what we see in *Figure 2.6* through the following steps:

1. Module `b.js` will link the exports from `a.js`, referring to them as `aModule`.
2. In turn, `a.js` will link to all the exports from `b.js`, referring to them as `bModule`.
3. Finally, `main.js` will import all the exports in `b.js`, referring to them as `b`; similarly, it will import everything from `a.js`, referring to them as `a`.

   Again, it's important to note that all the values are still uninitialized. In this phase, we are only linking references to values that will be available at the end of the next phase.

Phase 3—evaluation

The last step is the evaluation phase. In this phase, all the code in every file is finally executed. The execution order is again bottom-up, respecting the post-order depth-first visit of our original dependency graph. With this approach, `b.js` is executed first, then `a.js`, and `main.js` is the last file to be executed. This way, we can be sure that all the exported values have been initialized before we start executing our main business logic:

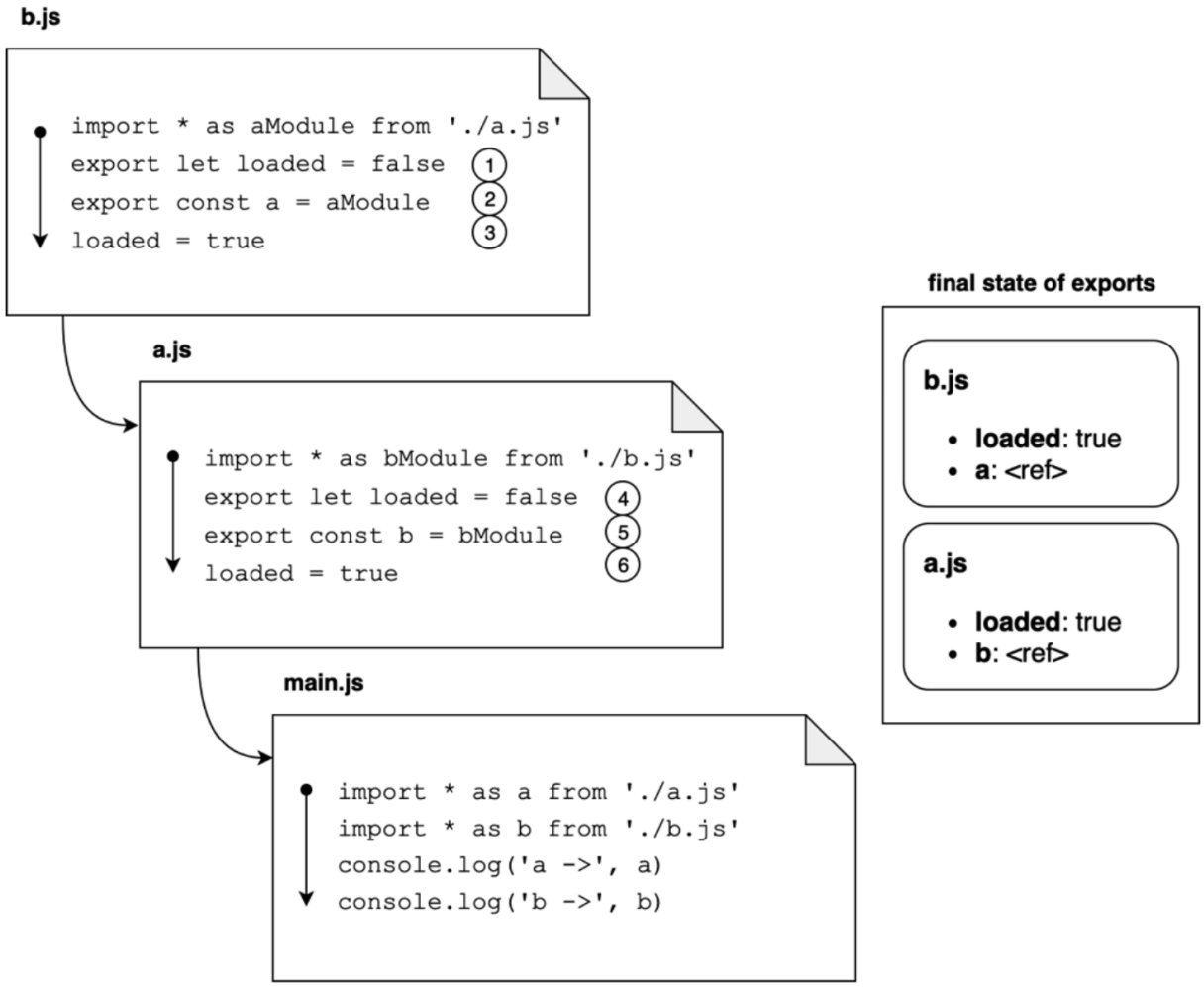

Figure 2.7: A visual representation of the evaluation phase

Following along from the diagram in *Figure 2.7*, this is what happens:

1. The execution starts from `b.js` and the first line to be evaluated initializes the `loaded` export to `false` for the module.
2. Similarly, the exported property `a` gets evaluated here. This time, it will be evaluated as a reference to the module object representing module `a.js`.
3. The value of the `loaded` property gets changed to `true`. At this point, we have fully evaluated the state of the exports for module `b.js`.
4. Now the execution moves to `a.js`. Again, we start by setting `loaded` to `false`.
5. At this point, the `b` export is evaluated to a reference to module `b.js`.
6. Finally, the `loaded` property is changed to `true`. Now we have finally evaluated all the exports for `a.js` as well.

After all these steps, the code in `main.js` can be executed, and at this point, all the exported properties are fully evaluated. Since imported modules are tracked as references, we can be sure every module has an up-to-date picture of the other modules, even in the presence of circular dependencies.

Modules that modify other modules

Sometimes, we come across modules that don't provide any exports. While this might seem unusual, it's important to recognize that a module can still offer functionality by modifying the global scope or objects within it, including other modules. Although this practice is generally discouraged, it can be useful and safe in specific situations, such as for testing, and is occasionally employed in real-world projects.This technique, where a module modifies other modules or objects in the global scope, is known as **monkey patching**. Monkey patching refers to the practice of altering existing objects at runtime to change or extend their behavior, or to apply temporary fixes.Let's consider an example of monkey patching. Suppose we have a module that provides a basic console logger implementation. We could then use another module, perhaps maintained by a third party, to enhance the original logger by adding ASCII color codes. This modification would colorize the logger's output in the terminal based on the log level: info messages in green, warnings in yellow, and errors in red. This colorizer module is optional, allowing users to enable colored output only if they choose to import it:

```
// logger.js
export const logger = {
  info(message) {
    console.log(`[INFO]\t${message}`)
  },
  error(message) {
    console.log(`[ERROR]\t${message}`)
  },
  warn(message) {
    console.log(`[WARN]\t${message}`)
  },
  debug(message) {
    console.log(`[DEBUG]\t${message}`)
  },
}
```

This is a basic logger implementation that exports a global `logger` object with methods such as `info()`, `error()`, `warn()`, and others. Each method prints a message with a prefix tag indicating the specific log level. This design helps users easily search and filter the logs produced by the application.Now, let’s implement the `colorizeLogger.js` module that enhances this logger by adding support for ASCII colors:

```
// colorizeLogger.js
import { logger } from './logger.js'
const RED = '\x1 b[31 m'
const YELLOW = '\x1 b[33 m'
const GREEN = '\x1 b[32 m'
const WHITE = '\x1 b[37 m'
const RESET = '\x1 b[0 m'
const originalInfo = logger.info
const originalWarn = logger.warn
const originalError = logger.error
const originalDebug = logger.debug
logger.info = message => originalInfo(`${GREEN}${message}${RESET}`)
logger.warn = message => originalWarn(`${YELLOW}${message}${RESET}`)
logger.error = message => originalError(`${RED}${message}${RESET}`)
logger.debug = message => originalDebug(`${WHITE}${message}${RESET}`)
```

`colorizeLogger.js` imports the original `logger.js` module and overrides the implementations of the `info`, `warn`, `error`, and `debug` methods. However, these new implementations still call the original methods to handle the actual logging. The `colorizeLogger.js` module only modifies the input message by applying ASCII color codes, while the original methods remain responsible for printing the log entries. This approach maintains a clear separation of concerns: the logger module is responsible for how logs are printed to the terminal, while the colorizer module is solely responsible for adding color to the messages before they are sent to the logger.Finally, let's see how we can use these two modules together in a hypothetical application:

```
// main.js
import { logger } from './logger.js'
import './colorizeLogger.js'
logger.info('Hello, World!')
logger.warn('Free disk space is running low')
logger.error('Failed to connect to database')
logger.debug('main() is starting')
```

In this file, importing `colorizeLogger.js` could be optional. You could comment it out to disable colored output without affecting the application's functionality. Additionally, since `colorizeLogger.js` does not export any entities, we use a simplified import syntax that omits the `from` keyword.This example illustrates how a module can patch another module using ES modules. However, this technique can be risky. Modifying the global namespace or other modules introduces **side effects**, impacting the state of entities outside their intended scope. This can lead to unpredictable consequences, especially when multiple modules interact with the same entities. For instance, if two different modules attempt to set the same global variable or alter the same property of a module,

the results can be uncertain (e.g., which module's changes prevail?). This unpredictability can affect the entire application.Therefore, it is important to use this technique with caution and ensure you understand all potential side effects before implementing it.

> *This logger implementation is purely for demonstration purposes and is not intended for production use. If you're looking for a logger for your Node.js applications, consider using* `pino` *(* `nodejsdp.link/pino` *), a highly efficient, comprehensive, and extensible logging library. Pino is designed with extensibility in mind, and there are extensions like* `pino-colada` *(* `nodejsdp.link/pino-colada` *) that provide enhanced output formatting and coloring.*

We learned that entities imported through ES modules are *read-only live bindings*, meaning we cannot reassign them from an external module. However, there is a caveat. While we can't change the bindings of the default export or named exports of an existing module from another module, if one of these bindings is an object, we can still mutate the object itself by reassigning some of its properties. This is precisely what we did in our logger example. Since `logger.js` exports a binding called `logger` that is an object, we are able to modify the properties within that object, though we wouldn't be able to reassign the logger binding entirely, as in the following example:

```
// replaceLogger.js
import { logger } from './logger.js'
const GREEN = '\x1 b[32 m'
// ...
const RESET = '\x1 b[0 m'
// replacing the logger binding with a new object
// this will fail with a TypeError: Assignment to constant variable.
logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // ...
}
```

This partially implemented example demonstrates a potential, but flawed, alternative approach to implementing our log output colorizer. Since this implementation attempts to override the `logger` binding entirely, running this module will lead to an error: `TypeError: Assignment to constant variable`. It's important to note, though, that while we cannot reassign the `logger` object itself, we could still patch the individual methods of the `logger` instance.An alternative (though still flawed) approach to monkey patching the entire `logger` binding could be as follows:

```
// replaceLogger2.js
import * as loggerModule from './logger.js'
const GREEN = '\x1 b[32 m'
// ...
const RESET = '\x1 b[0 m'
// replacing the logger binding inside the loggerModule with a new object
// this will fail with a TypeError: Cannot assign to read only property
loggerModule.logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // ...
}
```

The key difference in this approach is that we are now importing the entire `logger.js` module using a namespace import and then attempting to reassign the `logger` member within it. However, this approach also fails because module members are read-only bindings. Running this code would result in an error message confirming the issue:
`TypeError: Cannot assign to read only property 'logger' of object '[object Mo`
. Note that even in this case, while we cannot reassign the `logger` object itself, we could still patch the individual methods of the `logger` instance.There's another approach we could attempt, but it requires making some modifications to the logger.js module first:

```
// logger.js
export const logger = {
  // …
}
export default {
  logger,
}
```

We haven't changed the module's implementation, but we've added a default export that wraps all the module's members—specifically, the `logger` binding—into an object. By doing this, the default export becomes an object, and while we cannot reassign the entire default binding, we can still modify the individual members of the object. This approach provides the flexibility needed to create a functional alternative for our logger replacement module:

```
// replaceLogger3.js
import loggerModule from './logger.js'
```

```
const GREEN = '\x1 b[32 m'
// …
const RESET = '\x1 b[0 m'
loggerModule.logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // …
}
```

This approach closely resembles what we did with our implementation of the `replaceLogger2.js` module, but with one crucial difference: instead of performing a namespace import, we are now importing the default export from the `logger.js` module. With a namespace import, we receive a module instance with read-only members, preventing us from reassigning the `logger` member. By using a default import, however, we obtain a plain object. This allows us to modify its members, including the `logger` member, as intended. As a result, executing this code will proceed smoothly without errors.Let's use this new monkey-patching module. However, there's a caveat. Let’s say that we want to use the following code:

```
// main2 Broken.js
import { logger } from './logger.js'
import './replaceLogger3.js'
logger.info('Hello, World!')
```

This will mean that the output will not be colorized as expected. This is because we're importing the named export `logger` from `logger.js`, but `replaceLogger3.js` only patches the object exported as default export by `logger.js`. We can illustrate the effect of monkey patching applied by `replaceLogger3.js` in *Figure 2.8*:

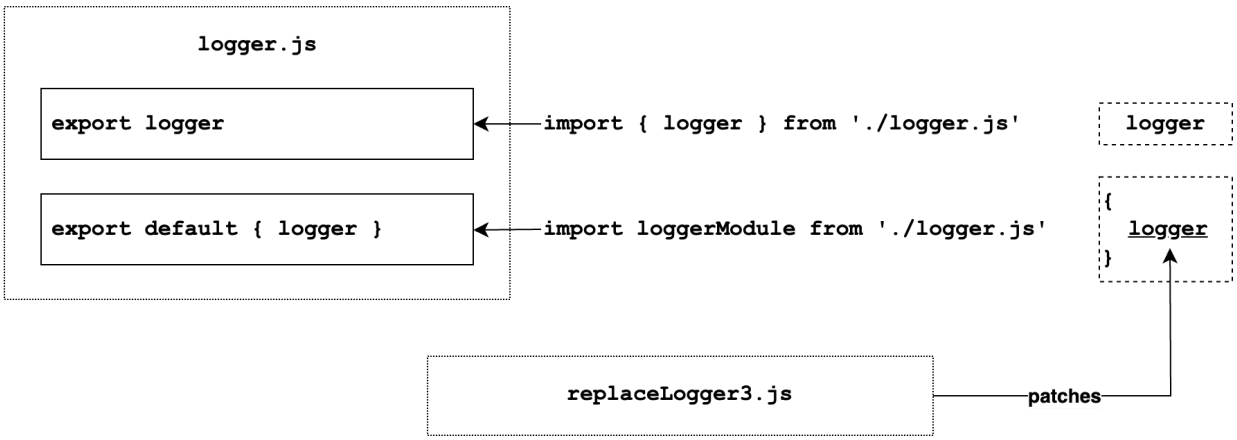

Figure 2.8: How replaceLogger3.js monkey-patches a member of the default export

In this example, `replaceLogger3.js` imports the object that `logger.js` exports as the default export. It then patches the specific `logger` binding within that object. This means the patching only affects the default export and not the named export. Therefore, to use the patched version of the logger, we must ensure that our main module imports the default export.Let's correct our code to do exactly this:

```
// main2.js
import loggerModule from './logger.js'
import './replaceLogger3.js'
loggerModule.logger.info('Hello, World!')
loggerModule.logger.warn('Free disk space is running low')
loggerModule.logger.error('Failed to connect to database')
loggerModule.logger.debug('main() is starting')
```

In this updated version, we import the default export from `logger.js`. By doing so, `replaceLogger3.js` patches the same reference that is used throughout the rest of the code, resulting in colorized output as intended.This approach is somewhat fragile because it requires several conditions to be met:

- The subject module (`logger.js` in our example) must export an object containing all its members as the default export.
- The patching module (`replaceLogger3.js` in our example) must import the default export and modify its members as needed.
- The using module (`main2.js` in our example) must import and use the default export of the subject module rather than its named exports directly.

If the subject or patching module is from a third party, we might not have control over these conditions, which could prevent us from applying this technique. However, as we will see in *Chapter 10*, *Testing*, this technique can be used with Node.js core modules and can be useful for writing unit tests. We will explore examples of using this patching technique to mock the filesystem module in unit tests.

> *Monkey patching can sometimes be used as an attack vector. For example, if an attacker manages to include malicious code in an application, this code might apply monkey patching to specific modules to be able to exfiltrate sensitive data, perform privilege escalation, or change important behaviors of the application. For this reason, the Node.js team has associated monkey patching with the CWE-349. Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weaknesses that can become vulnerabilities. The Node.js teams also provide some suggestions on how you might prevent monkey patching entirely if you want to harden your application:*
>
> *https://nodejsdp.link/cwe-349*

How monkey patching affects type safety in TypeScript projects

If you use TypeScript, you should be aware that applying monkey patching with TypeScript comes with additional challenges that need to be carefully considered. Before we explore some of these, keep in mind that TypeScript executes type checks and compiles your code to plain JavaScript before it can be executed. Monkey patching is something that happens at runtime, meaning it takes place only after TypeScript has done its job.Let's see what kind of challenges this creates:

- **Type safety issues**: TypeScript is built to provide type safety by ensuring that variables, functions, and objects are used according to their defined types. With monkey patching, you are generally altering the structure or behavior of existing objects or modules at runtime. You might be changing some types in a way that conflicts with the original type definitions. Imagine, for example, inadvertently changing the type of a member from a number to a string. This can lead to unexpected type errors at runtime. If, at some point in the code, the `toFixed()` method is called, this would result in a runtime error because the new type (`string`) does not have this particular method. TypeScript might be able to warn you about this potential issue if you are performing monkey patching in the same TypeScript project by providing some errors in the place where the patching is performed (typically with an error such as *"Type 'string' is not assignable to type 'number'"*). However, if the patching happens in a third-party library or a plain JavaScript file, in most circumstances, TypeScript won't be able to spot the type-safety issue and you'll be at risk of type-related runtime errors.
- **Challenges with type declaration files**: Working with TypeScript declaration files (`.d.ts`) becomes more complex when monkey patching is involved. You may need to extend existing types or modules, which can be tricky and error-prone, particularly with complex type structures.
- **Loss of IntelliSense and autocompletion**: One of TypeScript's major benefits is providing intelligent code completion and suggestions based on types. When you perform monkey patching, you might end up changing the signature of functions or the members of an object. The IDE might not be able to correctly capture these changes, so you might not see them being correctly represented when an autocompletion suggestion pops up. If you are adding extra functionality, for example, by adding a new method to a class, this method might not appear at all in the autocompletion box. All of these small issues can create development friction, slow down development, and increase the likelihood of mistakes.

Given these challenges, it's important to think carefully before using monkey patching in a TypeScript project. If you care about type safety, you should also be aware if you are importing third-party modules that might apply monkey patching as this might come with the undesired side effects that we just discussed. While monkey patching can be useful for testing, in most other cases, it should be viewed as a last resort for extending modules.

CommonJS modules

As we've already mentioned throughout this chapter, CommonJS is gradually being replaced by ES modules, and it's now recommended to use ES modules for new projects and libraries. However, CommonJS is not officially deprecated, and given its long-standing role in the JavaScript ecosystem, it's still important to have a basic understanding of it. This knowledge will enable you to work confidently with older code bases and libraries.Let's have a look at the basic syntax of CommonJS.Two of the main concepts of the CommonJS specification are as follows:

- `require, which` is a function that allows you to import a module from the local filesystem.
- `exports` and `module.exports, which` are special variables that can be used to export public functionality from the current module.

A simple CommonJS module might look like this:

```
// math.cjs
'use strict'
function add(a, b) {
  return a + b
}
module.exports = { add } // or `exports.add = add`
```

We can then use this module as follows:

```
'use strict'
const { add } = require('./math.cjs')
console.log(add(2, 3)) // 5
```

These snippets show how straightforward it is to export functionality using the `module.exports` variable (or simply `exports`) and how we can easily import that functionality in another module with the `require()` function.It's important to note that we explicitly enabled strict mode using `'use strict'`, as CommonJS does not operate in strict mode by default. We'll explore this detail further in the next section.Another crucial detail to note is that the `require()` function in CommonJS is both synchronous and dynamic. It operates in a straightforward manner—when executed, `require()` resolves the specified module, loads the associated file, and runs its content immediately, without needing a callback. This synchronous nature impacts how we define modules, as it generally limits us to using synchronous code during module initialization. This is also why many core Node.js libraries provide synchronous APIs alongside their asynchronous counterparts (e.g., `readFile` and `readFileSync` in `node:fs`).If a module requires asynchronous initialization, one approach is to define and export an uninitialized module that is later initialized asynchronously. However, this method doesn't ensure the module is ready for use immediately after being loaded. We will explore this issue and offer some elegant solutions in *Chapter 11*, *Advanced Recipes*.

Node.js originally included an asynchronous version of `require()`, but it was removed early on because it complicated a process meant for use during initialization, where asynchronous I/O often introduces more challenges than benefits.

ES modules and CommonJS—differences and interoperability

Let's now discuss some important differences between ES modules and CommonJS and how the two module systems can work together when necessary.

Strict mode

As opposed to CommonJS, ES modules run implicitly in strict mode. This means that we don't have to explicitly add the `'use strict'` statements at the beginning of every file. Strict mode cannot be disabled; therefore, we cannot use undeclared variables or the `with` statement or have other features that are only available in non-strict mode. However, this is definitely a good thing, as strict mode is a safer execution mode.

If you are curious to find out more about the differences between the two modes, you can check out a very detailed article on MDN Web Docs (https://nodejsdp.link/strict-mode).

Top-level await

Top-level await allows developers to use `await` at the top level of a module, simplifying asynchronous code without needing to wrap it in an `async` function. In ES modules, `await` can be used directly at the top level, as shown here:

```
// main.mjs
import { loadData } from 'someModule'
console.log(await loadData())
```

However, in CommonJS modules, `await` cannot be used at the top level directly. Instead, you must use it within an `async` function:

```
// main.cjs
'use strict'
const { loadData } = require('someModule')
async function main() {
  console.log(await loadData())
}
main()
```

If you try to use `await loadData()` outside of an async function, you'll get a `SyntaxError`. This limitation makes it easier to work with `async/await` in ES modules compared to CommonJS.

Behavior of `this`

An interesting difference between ES modules and CommonJS is the behavior of the `this` keyword.In the global scope of an ES module, `this` is `undefined`, while in CommonJS, `this` is a reference to `exports`:

```
// main.mjs - ES modules
console.log(this) // undefined
// main.cjs — CommonJS
console.log(this === exports) // true
```

> *Note that in both module systems, the behavior of the `globalThis` variable is consistent and will reference an object that contains global platform utilities such as `setInterval` or `structuredClone`. If you want to find out more about globalThis, check out https://nodejsdp.link/global-this.*

Missing references in ES modules

If you're accustomed to using CommonJS, you might be surprised by the absence of certain familiar references in ES modules, such as `require`, `exports`, `module.exports`, `__filename`, and `__dirname`. If we try to use any of them within an ES module, since it also runs in strict mode, we will get a `ReferenceError`:

```
console.log(exports) // ReferenceError: exports is not defined
console.log(module) // ReferenceError: module is not defined
console.log(__filename) // ReferenceError: __filename is not defined
console.log(__dirname) // ReferenceError: __dirname is not defined
```

`__filename` and `__dirname` represent the absolute path to the current module file and the absolute path to its parent folder. Those special variables can be very useful when we need to build a path relative to the current file.In ES modules, one useful object is `import.meta`. While we've already discussed `import.meta.resolve()`, there are other interesting properties within this object. For example, you can obtain equivalents to `__filename` and `__dirname` by using `import.meta.filename` and `import.meta.dirname`, respectively:

```
// main.js
const __filename = import.meta.filename // /path/to/project/main.js
const __dirname = import.meta.dirname // /path/to/project
```

These properties are still relatively new (they were introduced in Node.js v20.11.0, released in January 2024). If you're working with an earlier version, you can use `import.meta.url`, which is a reference to the current module file in a format similar to `file:///path/to/current_module.js`. Let's see how this value can be used to reconstruct the current file path and its parent directory in the form of absolute paths:

```
// main.js
import { fileURLToPath } from 'node:url'
import { dirname } from 'node:path'
const __filename = fileURLToPath(import.meta.url) // /path/to/project/main.
const __dirname = dirname(__filename) // /path/to/project
```

This works because `file.meta.url` will give us a URL representing the current file in the `"file:///path/to/project/main.js"` form. The utility `fileURLToPath()` takes a `"file://..."` URL and converts it to the equivalent path (`"/path/to/project/main.js"`). Finally, `dirname()` can take a path for a file and return the directory path for that file (`"/path/to/project"`).

Import interoperability

Sometimes, you may need to import an ES module from a CommonJS module or vice versa. There are a few important details and caveats to consider when handling these cross-module imports. Let's take a quick look at what's possible and how to manage these imports under different circumstances.

Import CommonJS modules from ES modules

An `import` statement can be used in an ES module to load a CommonJS module.Let's see a quick example:

```
// someModule.cjs
'use strict'
module.exports = {
  someFeature: 'someFeature',
}
```

This CommonJS module exports an object with a property `someFeature` using `module.exports`. Now, let's see how we can import this module from an ES module:

```
// main.js
import someModule from './someModule.cjs'
console.log(someModule)
```

This works as expected and the output of this file will be as follows:

```
{ someFeature: 'someFeature' }
```

However, what if we only want to import `someFeature`? Let's try:

```
// main2.mjs
import { someFeature } from './someModule.cjs'
console.log(someFeature)
```

If we run this code, we will see a rather verbose error message:

```
import { someFeature } from './someModule.cjs'
         ^^^^^^^^^^^
SyntaxError: Named export 'someFeature' not found. The requested module './
CommonJS modules can always be imported via the default export, for example
import pkg from './someModule.cjs';
const { someFeature } = pkg;
```

The issue arises because `someFeature` wasn't exported directly using `exports.someFeature`, so it isn't available as a named export when imported from an ES module. If you can modify the CommonJS module, you could solve this by assigning `someFeature` to `exports.someFeature`. However, if you can't change the source module (for example, if it's a third-party module installed from npm), there are a couple of workarounds you can rely on.The first solution is the one recommended by the preceding error message. We can import the entire module (using the default import syntax) and then use **destructuring assignment** syntax to extract `someFeature` from the imported object:

```
// main3.mjs
import someModule from './someModule.cjs'
const { someFeature } = someModule // destructuring assignment
console.log(someFeature)
```

In the previous snippet, the destructuring assignment is a compact way to extract the property `someFeature` from `someModule` and assign it to a local variable called `someFeature`. It is practically equivalent to writing the following:

```
const someFeature = someModule.someFeature
```

The other approach is to recreate the `require()` function in ES modules by using the `createRequire()` function provided by the core Node.js module `node:module`:

```
// main4.mjs
import { createRequire } from 'node:module'
const require = createRequire(import.meta.url)
const { someFeature } = require('./someModule.cjs')
console.log(someFeature)
```

The `require()` function that we created in this example behaves exactly like `require()` behaves in a CommonJS module. Therefore, we can use it without being too worried about incompatibilities. We will see how this can be used to import JSON files in the context of ES modules as well later in this chapter.

Import ES modules from CommonJS

At the time of writing this book, when trying to import an ES module into a CommonJS module, you might encounter issues. Let's see an example:

```
// someModule.mjs
export const someFeature = 'someFeature'
// main.cjs
'use strict'
const { someFeature } = require('./someModule.mjs')
```

Depending on your Node.js version, when you run `main.cjs`, you might see the following error:

```
Error [ERR_REQUIRE_ESM]: require() of ES Module […]/someModule.mjs not supp
Instead change the require of […]/someModule.mjs to a dynamic import() whic
```

The error message is clear: you can't use `require()` to import an ES module in a CommonJS module. However, you can use a dynamic `import()` instead, which is supported in CommonJS modules. Here's how to update the example:

```
// main2.cjs
'use strict'
async function main() {
  const { someFeature } = await import('./someModule.mjs')
  console.log(someFeature)
```

```
}
main()
```

This code works as expected. When executed, `someModule.mjs` is dynamically imported, and `"someFeature"` is printed to the console. However, keep in mind that because dynamic imports are asynchronous, and you're working within a CommonJS module, you can't use top-level `await`. You must wrap your logic in an `async` function and then invoke that function, which adds some boilerplate.Fortunately, Node.js offers an experimental flag, `--experimental-require-module`, that allows direct imports of ES modules from CommonJS modules. This means our original implementation can work if you run the file like this:

```
node --experimental-require-module main.cjs
```

This feature simplifies interoperability between CommonJS and ES modules, making it easier for developers and library maintainers to transition to ES modules without worrying about CommonJS compatibility. By the time you read this, it's possible this flag has become stable, eliminating the need for workarounds entirely.

> *Note that, at the time of writing, the `--experimental-require-module` flag does not support ES modules using top-level await.*

Importing JSON files

JSON is a widely used data format in web development, and it's common to need to load data from a JSON file into your programs. In CommonJS, this is straightforward:

```
// main.cjs
'use strict'
const data = require('./sample.json')
console.log(data)
```

Using `require()`, you can load and parse a JSON file with a single line of code. This function automatically reads the file's content, parses it as JSON, and returns the result. It's equivalent to manually reading the file with `readFileSync()` and parsing it with `JSON.parse()`, but with much less boilerplate.Now, let's see how to do the same in an ES module:

```
// main.mjs
import data from './sample.json'
console.log(data)
```

This seems logical, but it won't work. Running this code will result in an error:

```
TypeError [ERR_IMPORT_ATTRIBUTE_MISSING]: Module "[…]/sample.json" needs ar
```

This error is actually quite informative. It tells us that we need to specify that we're importing a JSON file by adding an **import attribute**. Here's how to do it:

```
// main2.mjs
import data from './sample.json' with { type: 'json' }
console.log(data)
```

This works, but at the time of writing, you'll see a warning that JSON module support is still experimental in Node.js.

> *The reason for this special syntax is security. By explicitly declaring the type, the ES module loader understands that it's dealing with JSON and won't try to execute any code from the module. This prevents a scenario where, even if the file has a* `.json` *extension, it could still contain JavaScript code that could then be executed, creating a potential attack vector. If you want to know more about this feature, you should check out the import attributes proposal repository at https://nodejsdp.link/proposal-import-attributes.*

This syntax, with some small syntactic differences, can also be used with dynamic ES module imports:

```
// main3.mjs
const { default: data } = await import('./sample.json', {
  with: { type: 'json' },
})
console.log(data)
```

Here, the only difference is that the result of the import is wrapped in a default property, so you need to access the JSON data through it. Like the static import, this code will also trigger the experimental feature warning.If you're not comfortable using experimental features, there are alternative approaches. The most obvious is to manually read and parse the JSON file:

```
// main4.js
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'
const jsonPath = join(import.meta.dirname, 'sample.json')
try {
  const dataRaw = await readFile(jsonPath, 'utf-8')
```

```
  const data = JSON.parse(dataRaw)
  console.log(data)
} catch (error) {
  console.error(error)
}
```

While this method works, it requires more code. A more concise option is to use the `require()` function within an ES module by utilizing the `createRequire()` utility that we discussed earlier:

```
// main5.mjs
import { createRequire } from 'node:module'
const require = createRequire(import.meta.url)
const data = require('./sample.json')
console.log(data)
```

Importing JSON files in Node.js can be straightforward or a bit tricky, depending on whether you're using CommonJS or ES modules. CommonJS makes it easy with the `require()` function, while ES modules add a bit more complexity with a special syntax to ensure security. Although the new syntax for importing JSON in ES modules is still experimental, it's designed to protect against potential risks. Whether you prefer the simplicity of CommonJS or want to embrace the security features of ES modules, you have options to choose from based on what works best for your project.

Using modules in TypeScript

When using different module systems in TypeScript, it's important to understand that TypeScript is a superset of JavaScript, designed to integrate smoothly with various ecosystems and platforms, including browsers, Node.js, and other JavaScript environments. As a compiler (or transpiler), TypeScript handles modules in two main contexts: how you structure your code with modules during development (input or *detected modules*) and the format of modules when your TypeScript code is compiled into JavaScript (output or *emitted modules*). TypeScript can even convert between different module systems, allowing you to write code using ES modules and compile it to CommonJS, for example.The way TypeScript operates can vary from project to project and is determined by the `tsconfig.json` configuration file. This file offers a wide range of options with considerable flexibility, which can sometimes be overwhelming and make it difficult to achieve the desired results.In this section, we will understand how TypeScript works when it comes to handling modules and cover some of the most important configuration options related to them. These options should help you create a configuration that meets your needs or, if you are working on an existing TypeScript code base, understand how modules are intended to be used in that particular project.

The role of the TypeScript compiler

The primary goal of the TypeScript compiler is to catch potential runtime errors during compile time. Whether or not modules are involved, the compiler must understand the specific characteristics of the expected runtime environment (the host system), including, for example, what global variables will be available. When modules are introduced, the compiler has to deal with additional challenges. Let's discuss them with an example:

```
// hello.ts
import sayHello from 'greetings'
sayHello('world')
```

To accurately compile `hello.ts`, the TypeScript compiler needs to determine several key factors about the structure of the input code and the characteristics of the target environment that will be executing the compiled code:

- **Module loading**: Assuming that the `greetings` module was originally written in TypeScript, will the module system load a TypeScript file (e.g., `greetings.ts`), or will it load a pre-compiles JavaScript file (e.g., `greetings.js`) that might be available in the same package?
- **Module type** and **module resolution**: What kind of module format does the target system expect, based on the file name and location? This is convention-based. In our example, the module specifier is only `"greetings"`, so this probably refers to a third-party library installed in the `node_modules` folder. However, this information alone doesn't explicitly dictate which exact file needs to be loaded. TypeScript also supports **path aliases** (or **path remapping**), which allows developers to define their own custom path prefixes to import files from the current project (https://nodejsdp.link/ts-path-aliases). This can be a convenient feature, but it's also something else that can affect module resolution. Therefore, it needs to be properly configured and accounted for during the compilation phase. Once a module file has been identified and loaded, the next question is: what module type is the loaded file using? We are using the ES module syntax when importing the `greetings` module, but this doesn't necessarily imply that the loaded file is an ES module. In fact, it might be a CommonJS module.
- **Output transformation**: How will the module syntax be transformed during the output process? For example, should all imports and exports be converted from ES modules to CommonJS?
- **Compatibility**: Can the detected module types interact correctly based on the syntax transformation?
- **Binding**: What specific export from the `greetings` module is bound to `sayHello`?

These questions are heavily dependent on the characteristics of the host system that consumes the output JavaScript or raw TypeScript—typically a runtime like Node.js or a bundler like Webpack. While the ECMAScript specification outlines how ES module imports and exports should link, it doesn't define how module resolution occurs or address other module systems like CommonJS. This means that runtimes and bundlers have significant leeway in designing their own rules. As a result, TypeScript's approach to these questions can vary depending on

where the code will run. There isn't a single correct answer, so the compiler needs to be configured appropriately.Thus, TypeScript's responsibility when it comes to modules can be summarized as follows:

- **Adapting to the host**: Understanding the rules of the host system (e.g. Node.js) well enough to compile files into a valid output module format.
- **Ensuring compatibility**: Ensuring that imports in the output files will resolve correctly.
- **Type assignment**: Assigning accurate types to imported entities.

> *For more details and examples, you can consult the official* **Modules (theory)** *section from the TypeScript website at https://nodejsdp.link/ts-modules-theory.*

Configuring the module output format

The `module` compiler option informs the compiler about the desired module format for emitted JavaScript. While its main role is to control the output format, it also guides the compiler on how to detect module types, manage imports between different module kinds, and handle features like `import.meta` and top-level `await`. Even if your TypeScript project doesn't emit JavaScript files (`noEmit`), selecting the right `module` setting is crucial for proper type-checking and IntelliSense.

> *For exhaustive guidance on choosing the right module setting for your project, see the* **Modules** *section in the official TypeScript handbook:*
>
> *https://nodejsdp.link/ts-modules*

When working with Node.js, we recommend setting the `module` option to `NodeNext`, reflecting the latest Node.js module system.

Input module syntax and output emission

It's important to understand that the input module syntax in your TypeScript files doesn't always correspond directly to the output module syntax in JavaScript files. For example, a file with ES module imports might be emitted as ES modules or transformed into CommonJS, depending on the `module` compiler option and any relevant detection rules. This flexibility means that merely inspecting an input file isn't enough to determine whether it's an ES module or a CommonJS module—TypeScript can convert between module systems, a feature that, while powerful, can sometimes lead to unexpected results and interoperability issues.In TypeScript 5.0, the `verbatimModuleSyntax` option was introduced to help developers understand exactly how their import and export statements will be emitted. When enabled, this flag requires that imports and exports in input files be written in the form that will undergo the least transformation before emission. For example, if a file is emitted as ES

modules, its imports and exports must be written in ES module syntax. When targeting Node.js, we recommend enabling `verbatimModuleSyntax` to keep results consistent and predictable.

Module resolution

While the ECMAScript specification defines the parsing and interpretation of import and export statements, it leaves the details of module resolution to the host system. To ensure that TypeScript interprets these statements in a way that is compatible with Node.js, you should set the `moduleResolution` option in your `tsconfig.json` file. For Node.js, we recommend setting this option to `NodeNext`, which complements the `module` option and defaults to the same value if not specified. `NodeNext` is designed to support upcoming Node.js module resolution features.

> *Writing the perfect TypeScript configuration file for your project requires a decent understanding of your specific context and how the TypeScript compiler works. Hopefully, we have provided you with the basics here. If you are looking for ready-made configurations that you can use in different contexts, this repo can be a great resource to check out:*
>
> *https://nodejsdp.link/total-typescript-tsconfig*

Summary

In this chapter, we explored the need for modules in JavaScript and the evolution of module systems in Node.js, focusing on CommonJS and ES modules. We covered key concepts like named and default exports, static and dynamic imports, and the module resolution algorithm. We also discussed the implications of circular dependencies, module loading phases, and how modules can modify others.We then looked at the differences between CommonJS and ES modules, including interoperability challenges, strict mode, and missing references. Additionally, we addressed the impact of monkey patching on type safety in TypeScript projects.We examined how to import JSON files in both module systems and workarounds for their limitations. Finally, we discussed how to leverage ES modules when using TypeScript.With this knowledge, you're now equipped to use both ES modules and CommonJS effectively, laying the groundwork for the next chapter on asynchronous programming in JavaScript. Get ready to explore callbacks, events, and their patterns in depth!

3

Callbacks and Events

In *synchronous programming*, we think of code as a series of consecutive steps that work together to solve a specific problem. Each operation is blocking, meaning that one task must be completed before the next one can begin. This approach makes the code straightforward to read, understand, and debug.On the other hand, *asynchronous programming* operates differently. Certain operations, like reading from a file or making a network request, run "in the background." When we initiate an *asynchronous operation*, the next instruction executes immediately, even if the previous task hasn't finished yet. In this context, we need a way to be notified when the asynchronous operation completes so that we can continue our execution flow with the operation's result. The most fundamental way to handle this in Node.js is through **callbacks**—functions that are invoked by the runtime once an asynchronous operation is complete.Callbacks are the foundational building blocks upon which all other asynchronous mechanisms are built. Without callbacks, we wouldn't have **promises**, and, in turn, we wouldn't have **async/await**. Additionally, **streams** and **events** also rely on callbacks. For this reason, understanding how callbacks work is crucial.It’s easy to assume that callbacks and event emitters are somewhat outdated or deprecated in modern JavaScript and Node.js, but that’s far from the truth. Truly understanding callbacks and event emitters is essential to grasping how

promises and async/await work under the hood. So, don't skip this chapter — it's your first important step for mastering asynchronous programming in JavaScript and Node.js!In this chapter, you'll dive deep into the Node.js Callback pattern and explore what it means to write asynchronous code in practice. We'll cover conventions, patterns, and common pitfalls. By the end of this chapter, you'll have a solid grasp of the basics of the Callback pattern.You'll also be introduced to the **Observer** pattern, which is closely related to the Callback pattern. The Observer pattern, represented by the `EventEmitter` in Node.js, uses callbacks to handle multiple, heterogeneous events and is one of the most widely used components in Node.js programming.To summarize, this is what you will learn in this chapter:

- The Callback pattern, how it works, what conventions are used in Node.js, and how to deal with its most common pitfalls.
- The Observer pattern and how to implement it in Node.js using the `EventEmitter` class.
- Comparing the `EventEmitter` with callbacks and how to combine the two to get the best of both worlds.

The Callback pattern

Callbacks are the embodiment of the Reactor pattern's handlers (which we covered in Chapter 1), a core component of the Node.js architecture. Callbacks can be defined as functions triggered to handle the result of an operation — exactly what we need when working with asynchronous tasks.

In an asynchronous context, callbacks take the place of the typical `return` statement, which only works synchronously. JavaScript is particularly well-suited for callbacks, as functions are first-class objects. This allows them to be easily assigned to variables, passed as arguments, returned from other functions, or stored in data structures.Callbacks played a key role in shaping Node.js's distinctive programming style for many years. Despite their decline in popularity in modern user-facing code, they remain a fundamental and indispensable component of both the Node.js and JavaScript ecosystems. In fact, callbacks are the building block for constructing Promises and Async/Await, and are essential to understanding these higher-level abstractions and to using them correctly. Additionally, many advanced techniques that will be explored in this chapter and the following ones require a solid understanding of callbacks. This is why we have chosen to start our exploration of mastering asynchronous patterns from the perspective of callbacks in this book.**Closures** also make an ideal partner for callbacks. They allow a function to retain access to the environment in which it was created, ensuring that the context of the asynchronous operation is preserved — no matter when or where the callback is eventually invoked.

> *If you need to refresh your knowledge about closures, you can refer to the article on MDN Web Docs at https://nodejsdp.link/mdn-closures.*

In this section, we'll take a closer look at this programming style that relies on callbacks rather than traditional `return` statements.

The continuation-passing style

In JavaScript (and therefore in Node.js), a callback is a function that is passed as an argument to a function *f*. This callback function will be eventually called with the result of the operation when *f* completes. In functional programming, this way of propagating the result is called **continuation-passing style** (**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 CPS

To clarify the concept, let's start with a simple synchronous function:

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

Nothing fancy here: just a simple addition function that takes two numbers and returns their sum. The result is returned to the caller using a `return` statement. This is known as **direct style** and is the most common way to return results in synchronous programming in most programming languages (including JavaScript).Now, here's the equivalent function using Continuation-Passing Style (CPS):

```
function addCps(a, b, callback) {
  callback(a + b)
}
```

If we compare the `addCps()` function with the `add()` function described before we can see 2 main differences:

- `addCps()` expects an additional argument: a `callback` function
- rather than using a `return` statement, `addCps()` propagates the result of the operation by invoking the `callback` function and by passing the result of the operation `(a + b)` as an argument.

The `addCps()` function is a synchronous CPS function. It's still synchronous because nothing asynchronous is happening during its execution. Once `addCps()` is called, `a + b` is calculated immediately and the `callback` is immediately invoked. Furthermore, `addCps()` completes its execution only when the callback does.In more general terms, the idea of CPS is that when we call a function, we also pass a *callback* function. Once the computation is finished, the callback is invoked with the result. In a sense, we're telling the function to "pass the ball" to the callback when it's done. In this particular example, since this operation is synchronous, the callback is called immediately. Let's clarify this further with an example:

```
console.log('before') // 1
addCps(1, 2, result => console.log(`Result: ${res
console.log('after') // 3
```

The previous code will print the following:

```
before
Result: 3
after
```

Let's try to make sense of this output:

1. the first line of code is performing a synchronous call that logs the word "*before*" in the standard output.
2. In the second line we invoke `addCps()` and we pass `1`, `2` and a callback to it. The callback function is an arrow function defined inline. If we check again the implementation of `addCps()`, we can see that the callback function will be immediately invoked with the result of the addition between our first 2 arguments (`1 + 2 = 3`). The body of the callback function therefore logs the string *"Result: 3"*.
3. In the last line, we simply log the word *"after"*.

At this point, you might still feel not 100% convince that everything that is happening here is synchronous. For example, you might be wondering: how do we know that `console.log()` is a synchronous function?This is a

legitimate question, and we could answer it by checking the documentation for this function, but, in general, as we explore more synchronous and asynchronous functions, we will learn to recognize common patterns and idioms and develop an instinct for when code is executed synchronously or asynchronously.So, let's see how *asynchronous* CPS works.

Asynchronous CPS

Now, let's look at an alternative implementation of `addCps()` that behaves asynchronously. Let's call it `addAsync()`:

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

In this example, we introduce an artificial delay using `setTimeout()` to simulate asynchronous behavior. The `setTimeout()` function schedules a task in the event queue to run after a specified number of milliseconds, making this an asynchronous operation. While this is a simple example meant to demonstrate the pattern, you can imagine real-world scenarios where the function might need to fetch external data, like a currency conversion function that retrieves the latest exchange rates before performing the conversion.

We have just learned that `setTimeout()` *is an asynchronous function, and by using it, we have transformed our* `addAsync()` *function into an asynchronous one as well. Asynchronous functions are often constructed by using lower-level asynchronous functions like* `setTimeout()`, `setInterval()`, `setImmediate()`, `process.nextTick()`, `fetch()`, `http.request()`, *and so on. The documentation for these functions can reveal their asynchronous nature, but as a general rule, whenever we are dealing with timers or waiting for input/output operations to complete (such as reading from a file or making an HTTP request), we will encounter some form of asynchronous abstraction.*

Let's try using `addAsync()` and observe how the order of operations changes:

```
console.log('before')
addAsync(1, 2, result => console.log(`Result: ${r
console.log('after')
```

The preceding code will print the following:

```
before
after
Result: 3
```

Since `setTimeout()` triggers an asynchronous operation, it doesn't wait for the callback to be executed; instead, it returns immediately, giving the control back to `addAsync()`, and then back again to its caller. This property in Node.js is crucial, as it gives control back to the event loop as soon as an asynchronous request is sent, thus allowing a new event from the queue to be processed.

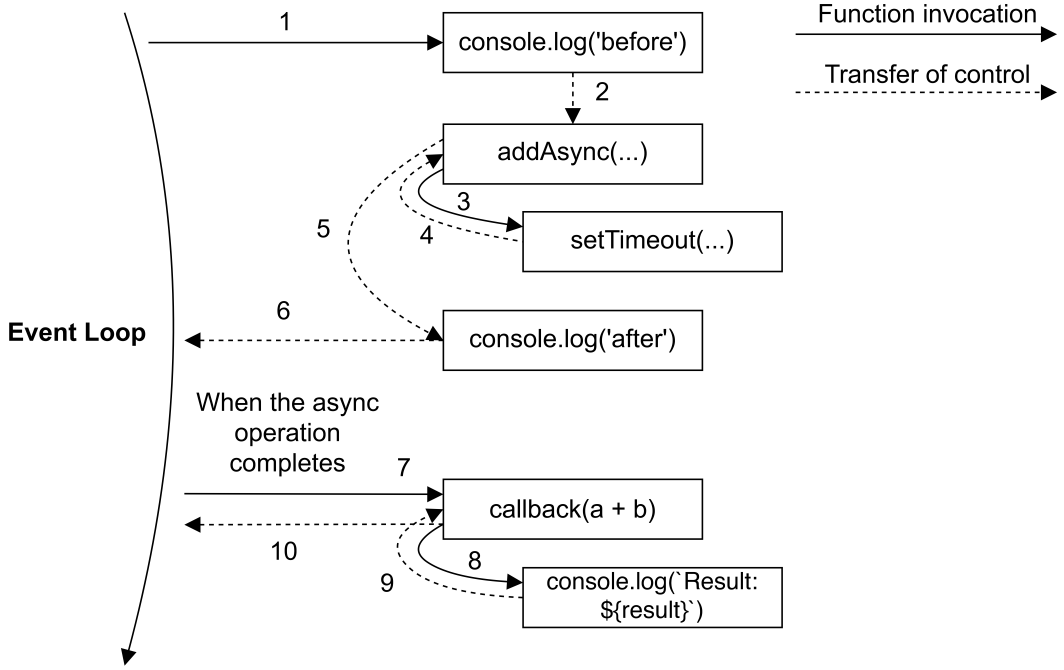

Figure 3.1: Control flow of an asynchronous function's invocation

Figure 3.1 shows step by step how this code is executed:

1. `console.log('before')` is executed: This logs the message "*before*" to the console.

2. `addAsync()` is executed: control is transferred to the `addAsync()` function, where two arguments and a callback are passed.
3. `setTimeout()` is invoked: Within `addAsync()`, the `setTimeout()` function schedules the callback to execute after a specified delay (in this case, 100 milliseconds).
4. Control is returned from `setTimeout()` to `addAsync()`: Because `setTimeout()` is asynchronous, it doesn't wait for the delay to complete before returning control.
5. Control returns to the calling function: Since `setTimeout()` is the last (and only) instruction in `addAsync()`, `addAsync()` returns. Control goes to the next line of code, `console.log('after')` which is immediately executed printing *"after"*.
6. Control goes to the event loop: There are no more instructions to execute, so control goes back to the event loop. The event loop is aware that there are pending tasks (the timer associated to the `setTimeout()` call) so it's going to wait for the async operation to complete.
7. The asynchronous task is completed: When the timeout completes after 100 milliseconds, the callback passed to `setTimeout()` is invoked.
8. The callback is executed and it receives as argument the result of the expression `a + b`.
9. `console.log('Result: ${result}')` is invoked. This is the body of the provided callback function. We called the argument

`result`, so this will print *"Result: 3"* (since `a` is `1` and `b` is `2`) in the console.

10. Control returns to the event loop: After the callback finishes execution, control returns to the event loop, which is now ready to handle other pending tasks, or, if there are no other tasks, exit.

It's important to emphasize that when a callback is executed, it starts with a fresh call stack, as this happens through the event loop. This is where JavaScript truly excels. Thanks to closures, it's simple to retain the context of the original caller, even if the callback is invoked later or from a different part of the code. As we'll explore in the examples later in this chapter, this makes asynchronous CPS a highly effective programming style.

> *The call stack is the mechanism used by JavaScript's interpreter to keep track of the execution context. It makes it possible to know where the program currently is and what functions are being executed. When a function is called, it is pushed onto the call stack. If that function calls another function, it too is added to the stack. Once a function completes, it is removed from the stack, and the interpreter resumes execution at the previous level. When a callback function is executed via the event loop, it begins with a fresh call stack. This clean execution context is fundamental to JavaScript's asynchronous behavior. Closures make it possible to retain the original context, ensuring that data and variables remain accessible when the callback eventually runs. This ability to retain context makes closures indispensable for*

Continuation-Passing Style programming, where functions are passed as arguments and executed later, often asynchronously. Closures bridge the gap between the temporary nature of the call stack and the persistent data needed for callbacks.

Non-CPS callbacks

There are many situations where the presence of a callback might lead us to assume that a function is asynchronous or using CPS, but that's not always the case. Take the `map()` method of an array, for example:

```
const result = [1, 5, 7].map(element => element -
console.log(result) // [0, 4, 6]
```

Here, the callback is simply used to iterate over the array elements, not to handle the result of the operation. In fact, the result is returned synchronously using direct style. There's no syntactic distinction between non-CPS callbacks and CPS ones, which is why the intended use of a callback should be clearly explained in the API documentation.In the next section, we'll dive into one of the most common pitfalls with callbacks that every Node.js developer needs to be aware of.

Synchronous or asynchronous?

You have seen how the execution 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 terms of 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 that might be very hard to detect and reproduce. To drive our analysis, we will take, as an example, the case of an inconsistently asynchronous function.

Writing an unpredictable function

One of the most dangerous situations is to have an API that behaves synchronously under certain conditions and asynchronously under others. Imagine a common scenario where we want to create an abstraction that fetches the content of a file from disk and caches it for faster future reads. Here's a potential (but flawed) implementation:

```
import { readFile } from 'node:fs'
const cache = new Map()
function inconsistentRead(filename, cb) {
  if (cache.has(filename)) {
    // invoked synchronously
    cb(cache.get(filename))
  } else {
    // asynchronous function
```

```
      readFile(filename, 'utf8', (_err, data) => {
        cache.set(filename, data)
        cb(data)
      })
    }
  }
```

This function uses the `cache` variable to store file contents. It's a very basic example with a few intrinsic issues — it lacks error handling (e.g. what happens if we fail to read the file), and the caching logic could be significantly improved (in *Chapter 11*, *Advanced Recipes*, you'll learn how to handle asynchronous caching properly). However, the real issue lies in its inconsistency: the function is asynchronous the first time a file is read but becomes synchronous once the file is cached. This unpredictable behavior can lead to unexpected problems, creating a recipe for all sorts of chaos! Let's discover how things can go terribly wrong with this approach!

Unleashing Zalgo

Now, let's discuss how the use of an unpredictable function, i.e. functions that can behave both synchronously and asynchronously every time we call them, such as the `inconsistentRead()` function that we previously defined above. These functions can break an application in non-obvious ways. Let's consider the following example to see some unexpected side effect that can happen when using the `inconsistentRead()` function:

```
function createFileReader(filename) {
  const listeners = []
  inconsistentRead(filename, (value) => {
    for (const listener of listeners) {
      listener(value)
    }
  })
  return {
    onDataReady: listener => listeners.push(liste
  }
}
```

When the preceding function is invoked, it creates a new object that acts as a notifier, allowing us 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 see how to use the `createFileReader()` function assuming we want to read a `data.txt` file in the current working directory which contains the text `some data`:

```
const reader1 = createFileReader('data.txt')
reader1.onDataReady(data => {
  console.log(`First call data: ${data}`)
  // ...sometime later we try to read again from
  // the same file
```

```
    const reader2 = createFileReader('data.txt')
    reader2.onDataReady(data => {
      console.log(`Second call data: ${data}`)
    })
  })
```

The preceding code will print the following text:

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

… And nothing else! What happened to our second `console.log()`? Why don't we see that in our output? Let's review the code more carefully to find out why:

- During the creation of `reader1`, our `inconsistentRead()` function behaves asynchronously because there is no cached result available for `data.txt`. This means that any `onDataReady` listener will be invoked later in another cycle of the event loop, so we have all the time we need to register our listener.
- Then, `reader2` is created in a cycle of the event loop in which the cache for `data.txt` already exists. In this case, the inner call to `inconsistentRead()` will be synchronous. So, its callback will be invoked immediately, which means that all the listeners of `reader2` will be invoked synchronously as well. However, we are registering the listener after the creation of `reader2`, so it 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 an argument, and the amount of time taken to load the file.This type of inconsistent behavior can lead to bugs that be extremely complicated to identify and reproduce in a real application. Imagine using 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 can definitely be considered a nasty defect.Isaac Z. Schlueter, the creator of npm and former Node.js project lead, once compared using this kind of unpredictable function to *unleashing Zalgo* in one of his blog posts.Zalgo is an internet legend about a terrifying entity said to bring chaos, insanity, and even the end of the world. If you haven't heard of Zalgo, I encourage you to look it up — it's definitely worth the read!

> *You can find Isaac Z. Schlueter's original post at https://nodejsdp.link/unleashing-zalgo.*

Using synchronous APIs

The lesson we must learn from the unleashing Zalgo example is that it is imperative for an API to clearly define its nature: either synchronous or asynchronous — mixing the two can lead to all sort of trouble.One way to fix our `inconsistentRead()` function is by making it fully synchronous. This is doable because Node.js offers synchronous APIs for most

basic I/O operations. For instance, we can use `readFileSync()` function instead of its asynchronous version. Here's the updated code:

```
import { readFileSync } from 'node:fs'
const cache = new Map()
function consistentReadSync(filename) {
  if (cache.has(filename)) {
    return cache.get(filename)
  }
  const data = readFileSync(filename, 'utf8')
  cache.set(filename, data)
  return data
}
```

You'll notice the function now uses direct style, meaning it returns data directly instead of using callbacks. Since it's synchronous, there's no need for a callback pattern. In fact, it's best practice to use direct style for synchronous APIs to keep things simple and clear.

Pattern

Always choose a direct style for purely synchronous functions

Keep in mind that switching an API from callback style (CPS) to direct style, or from asynchronous to synchronous (or vice

versa), is a breaking change. This means any code using the API will also need to be updated.

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

- A synchronous API for a specific functionality might not always be available.
- A synchronous API will block the event loop and put any concurrent requests on hold. This conflicts with the Node.js concurrency model, slowing down the whole application. We will discuss this in greater detail in *Chapter 11*: Advanced Recipes.

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 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 generally discouraged, but in some cases, it can be the easiest and most efficient option. If you're building a web server that handles many concurrent requests, it's important to use asynchronous APIs to avoid blocking the event loop, ensuring all requests are processed concurrently. On the other hand, if you're creating an automation script run from the command line and don’t need concurrency, it’s

perfectly fine to use synchronous APIs—they can make the code simpler and easier to manage.Sometimes, you might even use a mix of both. For example, you may need to load a configuration file before starting a web server. In this case, using a synchronous API to load the file makes sense, but your request-handling logic should use asynchronous, non-blocking code as much as possible.Always evaluate your specific use case to choose the right approach.

Pattern

Use blocking APIs sparingly and only when they don't affect the ability of the application to handle concurrent asynchronous operations.

Guaranteeing asynchronicity with 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 it being run immediately in the same event loop cycle. In Node.js, this is possible with `process.nextTick()`, which defers the execution of a function after the currently running operation completes. Its functionality is very simple: it takes a callback as an argument and pushes it to 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 currently running operation yields

control back to the event loop.Let's apply this technique to fix the issues found in the `inconsistentRead()` function:

```
import { readFile } from 'node:fs'
const cache = new Map()
function consistentReadAsync(filename, callback)
  if (cache.has(filename)) {
    // deferred callback invocation
    process.nextTick(() => callback(cache.get(fi
  } else {
    // asynchronous function
    readFile(filename, 'utf8', (_err, data) => {
      cache.set(filename, data)
      callback(data)
    })
  }
}
```

Now, thanks to `process.nextTick()`, our function is guaranteed to invoke its callback asynchronously, under any circumstances. Try to use it instead of the `inconsistentRead()` function and verify that, indeed, Zalgo has been eradicated.

Pattern

> *You can guarantee that a callback is invoked asynchronously by deferring its execution using* `process.nextTick()`.

You might have seen another API for deferring code execution: `setImmediate()`. Although its name suggests that it runs callbacks immediately, this can be a bit misleading. In reality, `setImmediate()` gives callbacks lower priority than those scheduled with `process.nextTick()` or even `setTimeout(callback, 0)`. Yes, naming things is hard!Callbacks deferred with `process.nextTick()` are called **microtasks** and they are executed just after the current operation completes, even before any other I/O event is fired. With `setImmediate()`, on the other hand, the execution is queued in an event loop phase that comes after all I/O events have been processed.This means that callbacks scheduled with `process.nextTick()` will always run sooner, but in certain situations, like when using recursion, it can lead to **I/O starvation** — indefinitely delaying I/O callbacks. For example, if you recursively schedule callbacks inside a `process.nextTick()` callback, more callbacks will keep piling up in the queue. Since the event loop prioritizes this queue over other callbacks, it won't get a chance to process I/O event callbacks, like reading from a file. We can appreciate this concept in the following example:

```
import { readFile } from 'node:fs'
readFile('data.txt', 'utf8', (_err, data) => {
  console.log(`Data from file: ${data}`)
})
let scheduledNextTicks = 0
```

```
function recursiveNextTick() {
  if (scheduledNextTicks++ >= 1000) {
    return
  }
  console.log('Keeping the event loop busy')
  process.nextTick(() => recursiveNextTick())
}
recursiveNextTick()
```

In this example, we read the content of a file using `readFile()` and provide a callback to log the data once it's available. After that, we artificially keep the event loop busy by recursively scheduling 1000 tasks using `process.nextTick()`. Each task simply prints "*Keeping the event loop busy*" to the console.If we run this code, the output will look like this:

```
Keeping the event loop busy
Keeping the event loop busy
Keeping the event loop busy
…
Data from file: some data
```

We've shortened the output, but in reality, you'll see "*Keeping the event loop busy*" printed 1000 times before the file's content appears! This demonstrates that while `process.nextTick()` is useful for running high-priority callbacks, it can become problematic if used too frequently.We mentioned that another common way to schedule callbacks to run as soon

as possible is `setTimeout(callback, 0)`, which behaves somewhat like `setImmediate()`. However, in typical scenarios, callbacks scheduled with `setImmediate()` are executed after those scheduled with `setTimeout(callback, 0)`.To understand why, we need to look at how the event loop handles callbacks in different phases. For the events we're discussing, timers (`setTimeout()`) are executed before I/O callbacks, which in turn are processed before `setImmediate()` callbacks. We can borrow (with slight modification) a great example from the official Node.js website (https://nodejsdp.link/next-tick):

```
setImmediate(() => {
  console.log('setImmediate(cb)')
})
setTimeout(() => {
  console.log('setTimeout(cb, 0)')
}, 0)
process.nextTick(() => {
  console.log('process.nextTick(cb)')
})
console.log('Sync operation')
```

Take a moment to think about the output.Ready? Here's what you'll see:

```
Sync operation
process.nextTick(cb)
```

```
setTimeout(cb, 0)
setImmediate(cb)
```

This gives us a clear picture of the event loop's order of priority for different types of events.

> *If you still feel confused about how JavaScript and Node.js work under the hood, or if you are wondering how the stack, the event loop and the different queues are used at different times of the execution of a program, we'd recommend checking out this awesome interactive simulator: https://nodejsdp.link/next-tick* `nodejsdp.link/js-visualizer`.

The differences between these APIs will become even clearer later in the book when we discuss running synchronous, CPU-bound tasks (*Chapter 11, Advanced Recipes)*.Next, we'll explore the conventions for defining callbacks in Node.js.

Node.js callback conventions

In Node.js, CPS APIs and callbacks follow a set of specific conventions. These conventions aren't just found in the core library, most third-party modules and applications follow them too. To design an async API that uses callbacks effectively, it's crucial you understand these guidelines and stick to them.

The callback is the last argument

In all core Node.js functions, the standard convention is that when a function accepts a callback as input, this must be passed as the last argument.Let's take the following Node.js core API as an example:

```
readFile(filename, [options], callback)
```

As you can see from the signature of the preceding function, the callback is always put in the last position, even in the presence of optional arguments such as `options` in this example. The reason for this convention is that the function call is more readable in case the callback is defined in place.

Any error always comes first

In CPS, errors are propagated like any other type of result, which means using callbacks. In Node.js, any error produced by a CPS function is always passed as the first argument of the callback, and any actual result is passed starting from the second argument. If the operation succeeds without errors, the first argument will be `null` or `undefined`. The following code shows you how to define a callback that complies with this convention:

```
readFile('foo.txt', 'utf8', (err, data) => {
  if (err) {
    handleError(err)
  } else {
```

```
    processData(data)
  }
})
```

It is best practice to always check for the presence of an error, as not doing so will make it harder for you to debug your code and discover the possible points of failure. Another important convention to consider is that the error must always be an instance of the `Error` class. This means that simple strings or numbers should never be passed as error objects.

Propagating errors

Error propagation in synchronous, direct style functions is done with the well-known `throw` statement, which causes the error to jump up in the call stack until it is caught:

```
throw new Error('Something went wrong')
```

In asynchronous CPS, however, proper error propagation is done by simply passing the error to the next callback in the chain. The typical pattern looks as follows:

```
import { readFile } from 'node:fs'
function readJson(filename, callback) {
  readFile(filename, 'utf8', (err, data) => {
```

```
    let parsed
    if (err) {
      // error reading the file
      // propagate the error and exit the current
      return callback(err)
    }
    try {
      // parse the file contents
      parsed = JSON.parse(data)
    } catch (err) {
      // catch parsing errors
      return callback(err)
    }
    // no errors, propagate just the data
    callback(null, parsed)
  })
}
```

Notice how we don't throw or return errors. The first possible error can happen when we use the `readFile()` operation. Inside its callback, the first argument we receive is `err` and it represents an error related to reading from the given file. We do not throw it or return it; instead, we just invoke the `callback` with the error to propagate it back to the caller of `readJson()`. The `return` statement is used only to stop the function execution, not to return a specific value. The next potential error that we need to deal with is when we call `JSON.parse()`. This is a synchronous function and therefore, if we try to parse invalid JSON, it will use the tradi-

tional `throw` instruction to propagate errors to the caller. This means that this time we need to use a `try/catch` block to capture errors. Again, on the catch branch we invoke `callback(err)` to propagate the error to the caller and we use `return` to stop the execution of the function. Finally, if everything went well, `callback` is invoked with `null` as the first argument to indicate that there are no errors.It's also interesting to note how we refrained from invoking `callback` from within the `try` block. This is because doing so would catch any error thrown from the execution of the callback itself, which is usually not what we want.

Avoiding uncaught exceptions

Sometimes, it can happen that an error is thrown and not caught within the callback of an asynchronous function. This could happen if, for example, we forget to surround `JSON.parse()` with a `try...catch` and then our function happens to parse some invalid JSON at runtime.Throwing an error inside an asynchronous callback would cause the error to jump up to the event loop, so it would never be propagated to the next callback. In Node.js, this is an unrecoverable state and the application would simply exit with a non-zero exit code, printing the stack trace to the `stderr` interface.To see this in action, let's try to remove the `try...catch` block surrounding `JSON.parse()` from the `readJson()` function we defined previously:

```
function readJsonThrows(filename, callback) {
  readFile(filename, 'utf8', (err, data) => {
    if (err) {
      return callback(err)
    }
    callback(null, JSON.parse(data))
  })
}
```

Now, in the function we just defined, there is no way of catching an eventual exception coming from `JSON.parse()`. If we try to parse an invalid JSON file with the following code:

```
readJsonThrows('invalid_json.json', (err) => cons
```

Although we have provided a callback that should be able to handle the errors, since the implementation of `readJsonThrows()` does not call the callback but just throws an error, this will result in the application being abruptly terminated, with a stack trace similar to the following being printed on the console:

```
SyntaxError: Unexpected token h in JSON at positi
    at JSON.parse (<anonymous>)
    at file:///.../03-callbacks-and-events/10-unc
    at FSReqCallback.readFileAfterClose [as oncor
```

Now, if you look at the preceding stack trace from the bottom up, you will see that it starts from within the built-in `fs` module, and exactly from the point in which the native API has completed reading and returned its result back to the `fs.readFile()` function, via the event loop. This clearly shows that the exception traveled from our callback, up the call stack, and then straight into the event loop, where it was finally caught and thrown to the console.This also means that wrapping the invocation of `readJsonThrows()` with a `try...catch` block will not help us to catch the error, because the stack in which the block operates is different from the one in which our callback is invoked. The following code shows the anti-pattern that was just described:

```
try {
  readJsonThrows('invalid_json.json', (err) => co
} catch (err) {
  console.log('This will NOT catch the JSON parsi
}
```

The preceding `catch` statement will never receive the JSON parsing error, as it will travel up the call stack in which the error was thrown, that is, in the event loop and not in the function that triggered the asynchronous operation.As mentioned previously, the application will abort the moment an exception reaches the event loop. However, we still have the chance to perform some cleanup or logging before the application terminates. In fact, when this happens, Node.js will emit a special event called

`uncaughtException`, just before exiting the process. The following code shows a sample use case:

```
process.on('uncaughtException', (err) => {
  console.error(`This will catch at last the JSON
  // Terminates the application with 1 (error) as
  // Without the following line, the application
  process.exit(1)
})
```

It's important to understand that an uncaught exception leaves the application in a state that is not guaranteed to be consistent, which can lead to unforeseeable problems. For example, there might still be incomplete I/O requests running or closures might have become inconsistent. That's why it is always advised, especially in production, to never leave the application running after an uncaught exception is received. Instead, the process should exit immediately, optionally after having run some necessary cleanup tasks, and ideally, a supervising process should restart the application. This is also known as the **fail-fast** approach and it's the recommended practice in Node.js and it will be discussed in more detail in *Chapter 12*, *Scalability and Architectural Patterns*.This concludes our gentle introduction to the Callback pattern. Now, it's time to meet the Observer pattern, which is another critical component of an event-driven platform such as Node.js.

The Observer pattern

Another important and fundamental pattern used in Node.js is the **Observer** pattern. Together with the Reactor pattern and callbacks, the Observer pattern is an absolute requirement for mastering the asynchronous world of Node.js.The Observer pattern is the ideal solution for modeling the reactive nature of Node.js and a perfect complement for callbacks. Let's give a formal definition, as follows:The Observer pattern defines an object (called subject) that can notify a set of observers (or listeners) when a change in its state occurs.The main difference from the Callback pattern is that the subject can notify multiple observers, while a traditional CPS callback will usually propagate its result to only one listener, the callback.

The EventEmitter

In traditional object-oriented programming, the Observer pattern requires interfaces, concrete classes, and a hierarchy. In Node.js, all this becomes much simpler. The Observer pattern is already built into the core and is available through the `EventEmitter` class. The `EventEmitter` class allows us to register one or more functions as listeners, which will be invoked when a particular event type is fired. *Figure 3.2* visually explains this concept:

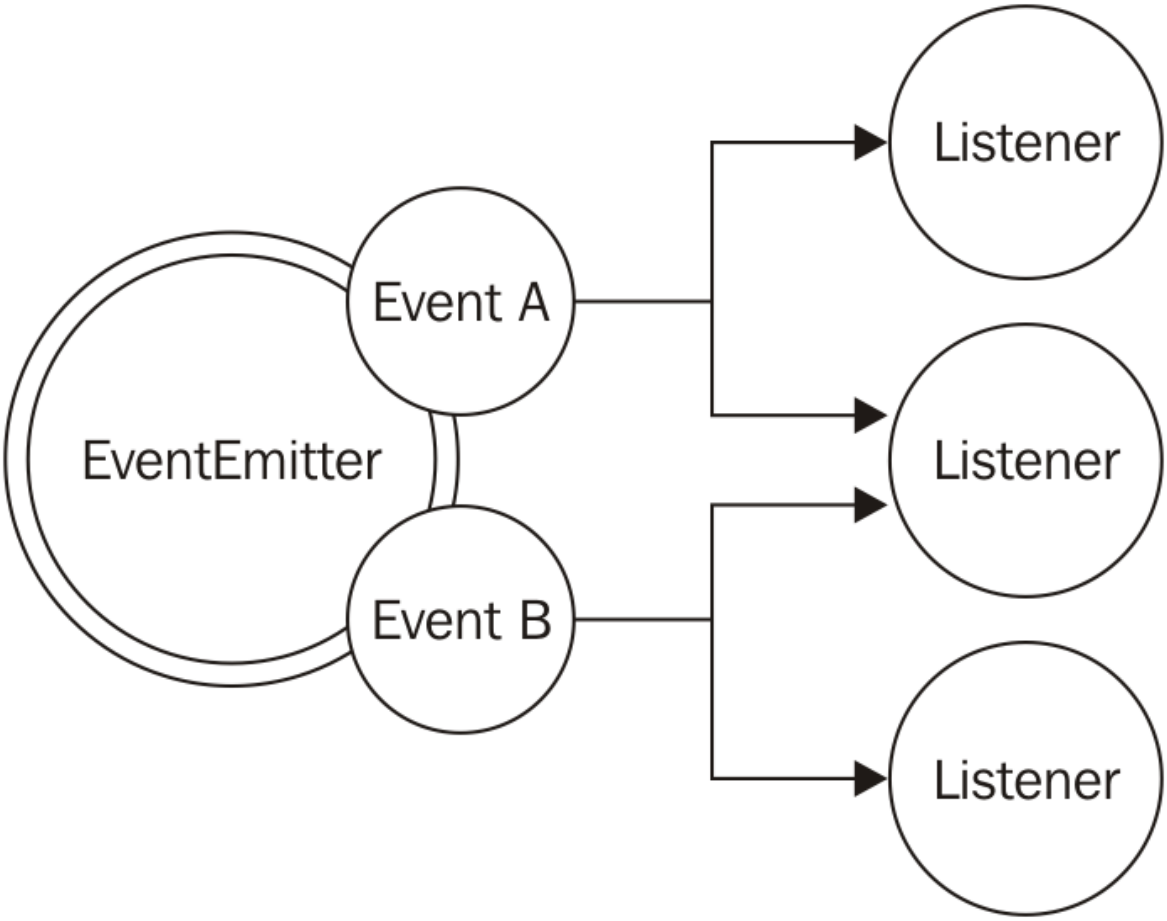

Figure 3.2: Listeners receiving events from an EventEmitter

The `EventEmitter` is exported from the `node:events` core module. The following code shows how we can obtain a reference to it:

```
import { EventEmitter } from 'node:events'
const emitter = new EventEmitter()
```

The essential methods of the `EventEmitter` are as follows:

- `on(event, listener)` : This method allows us to register a new listener (a function) for the given event type (a string).
- `once(event, listener)` : This method registers a new listener, which is then removed after the event is emitted for the first time.
- `emit(event, [arg1], [...])` : This method produces a new event and provides additional arguments to be passed to the listeners.
- `removeListener(event, listener)` : This method removes a listener for the specified event type. Note that this method has also an alias: `off(event, listener)` .
- `removeAllListeners(event)` : Removes all the listeners registered for the given event.

All these methods will return the same `EventEmitter` instance to allow chaining. The `listener` function has the signature `function([arg1], [...])` , so it simply accepts the arguments provided at the moment the event is emitted.You can already see that there is a big difference between a listener and a traditional Node.js callback. In fact, the first argument is not an error, but it can be any data passed to `emit()` at the moment of its invocation.

Creating and using the EventEmitter

Let's now see how we can use an `EventEmitter` in practice. The simplest way is to create a new instance and use it immediately. The following code shows us a function that uses an `EventEmitter` to notify its sub-

scribers in real time when a particular regular expression (regex) is matched in a list of files:

```
import { EventEmitter } from 'node:events'
import { readFile } from 'node:fs'
function findRegex(files, regex) {
  const emitter = new EventEmitter()
  for (const file of files) {
    readFile(file, 'utf8', (err, content) => {
      if (err) {
        return emitter.emit('error', err)
      }
      emitter.emit('fileread', file)
      const match = content.match(regex)
      if (match) {
        for (const elem of match) {
          emitter.emit('found', file, elem)
        }
      }
    })
  }
  return emitter
}
```

The function we just defined returns an `EventEmitter` instance that will produce three events:

- `fileread`, when a file is being read
- `found`, when a match has been found
- `error,` when an error occurs during reading the file

Let's now see how our `findRegex()` function can be used:

```
findRegex(
  ['fileA.txt', 'fileB.json'],
  /hello [\w.]+/
)
  .on('fileread', file => console.log(`${file} wa
  .on('found', (file, match) => console.log(`Matc
  .on('error', err => console.error(`Error emitte
```

In the code we just defined, we register a listener for each of the three event types produced by the `EventEmitter` that was created by our `findRegex()` function.

> *We kept this example intentionally simple to show how to use the `EventEmitter` class, but there's room for improvement, especially for production use. One issue is that we're loading all files into memory, which could lead to problems if the files are large, potentially filling up memory and crashing the program. In Chapter 6: Coding with Streams, we'll cover techniques to handle these situations more reliably. Additionally, we're using a simple synchronous `for...of` loop to read each file, triggering a `readFile()` op-*

eration for every file at once. This approach can overwhelm the system if there are many files to process, potentially exceeding system limits on open file descriptors. A better solution would be to process the files in batches using the limited parallel execution pattern, which we'll discuss in Chapter 4: Asynchronous Control Flow Patterns with Callbacks.

Propagating errors

As with callbacks, the `EventEmitter` can't just `throw` an exception when an error condition occurs. Instead, the convention is to emit a special event, called `error`, and pass an `Error` object as an argument. That's exactly what we were doing in the `findRegex()` function that we defined earlier.

The `EventEmitter` class treats the `error` event in a special way. It will automatically throw an exception and exit from the application if such an event is emitted and no associated listener is found. For this reason, it is recommended to always register a listener for the `error` event.

Making any object observable

In the Node.js world, the `EventEmitter` is rarely used on its own, as you saw in the previous example. Instead, it is more common to see it ex-

tended by other classes. In practice, this enables any class to inherit the capabilities of the `EventEmitter`, hence becoming an observable object.-
To demonstrate this pattern, let's try to implement the functionality of the `findRegex()` function in a class, as follows:

```
import { EventEmitter } from 'node:events'
import { readFile } from 'node:fs'
class FindRegex extends EventEmitter {
  constructor(regex) {
    super()
    this.regex = regex
    this.files = []
  }
  addFile(file) {
    this.files.push(file)
    return this
  }
  find() {
    for (const file of this.files) {
      readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err)
        }
        this.emit('fileread', file)
        const match = content.match(this.regex)
        if (match) {
          for (const elem of match) {
```

```
              this.emit('found', file, elem)
            }
          }
        })
      }
      return this
    }
  }
```

This code defines a `FindRegex` class that extends the `EventEmitter` class from the `node:events` module. This class has a constructor that uses `super()` to initialize the `EventEmitter` internals. It also takes in a `regex` argument and creates an array of files. The class has two methods: `addFile()` and `find()`. The `addFile()` method adds a file to the files array and returns the current instance of the `FindRegex` class to allow chained calls. The `find()` method reads each file in the files array, uses the `readFile()` function to read the contents of the file, and then uses the `match()` function to search for matches in the content using the regex passed as an argument. If a match is found, it emits the `found` event with the file path and the matched text.The following is an example of how to use the `FindRegex` class we just defined:

```
const findRegexInstance = new FindRegex(/hello [\
findRegexInstance
  .addFile('fileA.txt')
```

```
    .addFile('fileB.json')
    .find()
    .on('found', (file, match) => console.log(`Matc
    .on('error', err => console.error(`Error emitte
```

You will now notice how the `FindRegex` object also provides the `on()` method, which is inherited from the `EventEmitter`. This is a common pattern in the Node.js ecosystem. For example, the `Server` object of the core HTTP module inherits from the `EventEmitter` function, thus allowing it to produce events such as `request` (when a new request is received), `connection` (when a new connection is established), or `closed` (when the server socket is closed).Other notable examples of objects extending the `EventEmitter` are Node.js streams. We will analyze streams in more detail in *Chapter 6*, *Coding with Streams*.

The risk of memory leaks

A memory leak is a software defect whereby memory that is no longer needed is not released, causing the memory usage of an application to grow indefinitely.When subscribing to observables with a long life span, it is extremely important that we **unsubscribe** our listeners once they are no longer needed. This allows us to release the memory used by the objects in a listener's scope and prevent **memory leaks**. Unreleased `EventEmitter` listeners are the main source of memory leaks in Node.js (and JavaScript in general).For example, consider the following code:

```
const thisTakesMemory = 'A big string....'
const listener = () => {
  console.log(thisTakesMemory)
}
emitter.on('an_event', listener)
```

The variable `thisTakesMemory` is referenced in the listener and therefore its memory is retained until the listener is released from `emitter`, or until the `emitter` itself is garbage collected, which can only happen when there are no more active references to it, making it unreachable.

> *You can find a good explanation about garbage collection in JavaScript and the concept of reachability at https://nodejsdp.link/garbage-collection.*

This means that if an `EventEmitter` remains reachable for the entire duration of the application, all its listeners do too, and with them all the memory they reference. If, for example, we register a listener to a "permanent" `EventEmitter` at every incoming HTTP request and never release it, then we are causing a memory leak. The memory used by the application will grow indefinitely, sometimes slowly, sometimes faster, but eventually it will crash the application. This is something that, under specific circumstances, an attacker could be able to exploit to trigger a Denial of Service (DoS) attack. To prevent such a dangerous situation, we can release

the listener with the `removeListener()` method of the `EventEmitter`:

```
emitter.removeListener('an_event', listener)
```

An `EventEmitter` has a very simple built-in mechanism for warning the developer about possible memory leaks. When the count of listeners registered to an event exceeds a specific amount (by default, 10), the `EventEmitter` will produce a warning. Sometimes, registering more than 10 listeners is completely fine, so we can adjust this limit by using the `setMaxListeners()` method of the `EventEmitter`.

> *When we only want to listen to an event once, we can use the convenience method `once(event, listener)` in place of `on(event, listener)` to automatically unregister a listener after the event is received for the first time. However, be aware that if the event we specify is never emitted, then the listener is never released, causing a memory leak.*

Synchronous and asynchronous events

As with callbacks, events can also be emitted synchronously or asynchronously with respect to the moment the tasks that produce them are triggered. It is crucial that we never mix the two approaches in the same `EventEmitter`, but even more importantly, we should never emit the

same event type using a mix of synchronous and asynchronous code, to avoid producing the same problems described in the *Unleashing Zalgo* section. The main difference between emitting synchronous and asynchronous events lies in the way listeners can be registered.When events are emitted asynchronously, we can register new listeners, even after the task that produces the events is triggered, up until the current stack yields to the event loop. This is because the events are guaranteed not to be fired until the next cycle of the event loop, so we can be sure that we won't miss any events.The `FindRegex()` class we defined previously emits its events asynchronously after the `find()` method is invoked. This is the reason why we can register the listeners *after* the `find()` method is invoked, without losing any events, as shown in the following code:

```
findRegexInstance
  .addFile(...)
  .find()
  .on('found', ...)
```

On the other hand, if we emit our events synchronously after the task is launched, we have to register all the listeners *before* we launch the task, or we will miss all the events. To see how this works, let's modify the `FindRegex` class we defined previously and make the `find()` method synchronous:

```
find() {
  for (const file of this.files) {
    let content
    try {
      content = readFileSync(file, 'utf8')
    } catch (err) {
      this.emit('error', err)
    }
    this.emit('fileread', file)
    const match = content.match(this.regex)
    if (match) {
      for (const elem of match) {
        this.emit('found', file, elem)
      }
    }
  }
  return this
}
```

Now, let's try to register a listener before we launch the `find()` task, and then a second listener after that to see what happens:

```
const findRegexSyncInstance = new FindRegexSync(/
findRegexSyncInstance
  .addFile('fileA.txt')
  .addFile('fileB.json')
  // this listener is invoked
```

```
    .on('found', (file, match) => console.log(`[Bef
    .find()
    // this listener is never invoked
    .on('found', (file, match) => console.log(`[Aft
```

As expected, the listener that was registered after the invocation of the `find()` task is never called; in fact, the preceding code will print:

```
[Before] Matched "hello world"
[Before] Matched "hello Node.js"
```

There are some (rare) situations in which emitting an event in a synchronous fashion makes sense, but the very nature of the `EventEmitter` lies in its ability to deal with asynchronous events. Most of the time, emitting events synchronously is a telltale sign that we either don't need the `EventEmitter` at all or that, somewhere else, the same observable is emitting another event asynchronously, potentially causing a Zalgo type of situation.

> *The emission of synchronous events can be deferred with `process.nextTick()` to guarantee that they are emitted asynchronously.*

EventEmitter versus callbacks

When designing an asynchronous API, a common question is whether to use an `EventEmitter` or a callback. A simple guideline is to use callbacks for returning results asynchronously, and events when you want to notify that something has happened, leaving it up to the consumer to decide whether to handle that event.The confusion comes from the fact that both approaches are quite similar in practice and can often produce the same results. For instance, let's compare 2 different code examples.The first one illustrates the events API:

```
import { EventEmitter } from 'node:events'
function helloEvents () {
  const eventEmitter = new EventEmitter()
  setTimeout(() => eventEmitter.emit('complete',
  return eventEmitter
}
helloEvents().on('complete', message => console.l
```

The second one illustrates the usage of callbacks:

```
function helloCallback (cb) {
  setTimeout(() => cb(null, 'hello world'), 100)
}
helloCallback((_err, message) => console.log(mess
```

The two functions `helloEvents()` and `helloCallback()` can be considered equivalent in terms of functionality. They both use setTimeout() to simulate a delay before producing a result (the string *"hello world"* in this case). They differ in the way they signal the completion of the timeout. `helloEvents()` uses an event, while `helloCallback()` relies on a callback. The key differences between these two approaches lie in readability, semantics, and the amount of code required to implement or use them.While there are no strict rules for choosing one approach over the other, here are a few tips to guide your decision:

- Callbacks can be limiting when supporting different types of events. You can pass the event type as an argument to the callback or use multiple callbacks for each event, but this approach can result in a less elegant API. In such cases, an `EventEmitter` provides a cleaner interface and more concise code.
- Use `EventEmitter` when an event may happen multiple times or not at all. Callbacks are typically expected to run once, regardless of success or failure. If an event can recur, it's better treated as something to be communicated rather than a result to be returned.
- An `EventEmitter` allows multiple listeners to be registered for the same event. This flexibility can be useful in certain scenarios.

Combining callbacks and events

In some cases, you can use an `EventEmitter` alongside a callback. This pattern is very powerful and useful when you need to handle the final result of an asynchronous operation with a callback, but also want to emit progress events as the operation is ongoing. This approach gives you the best of both worlds, allowing for granular event tracking while still providing a way to propagate a final result.A traditional example is downloading a file from a URL. We can create a function that uses a callback to signal when the download is complete. At the same time, the function can return an `EventEmitter` to track the download's progress. This allows us to display real-time updates, such as the percentage of completion, on the screen.Before diving into the code, it's important to note that we'll be building an HTTPS client that follows a request/response flow. Here's a breakdown of the main steps:

1. **Client request**: The client sends a request to the server, specifying a method (in our case, GET) and a path (the part of the URL after the protocol and domain name). This is an asynchronous operation since multiple bytes are transmitted over the network gradually.
2. **Server response**: The server processes the request and sends back a response. This response typically has two parts: a response head (which includes the status code and headers) and a response body, which, in this case, will be the content of the file being downloaded. This is also an asynchronous operation.
3. **Closing the connection**: Once the response is fully received, the connection is closed on both ends, completing the flow. We also need to ac-

count for potential errors, such as failing to establish a connection, or the server dropping the response midway through the transfer.

Now let's how we can implementat this concept:

```
import { EventEmitter } from 'node:events'
import { get } from 'node:https'
function download(url, cb) { // 1
  const eventEmitter = new EventEmitter() // 2
  const req = get(url, resp => { // 3
    const chunks = [] // 4
    let downloadedBytes = 0
    const fileSize = Number.parseInt(
      resp.headers['content-length'], 10
    )
    resp
      .on('error', err => { // 5
        cb(err)
      })
      .on('data', chunk => { // 6
        chunks.push(chunk)
        downloadedBytes += chunk.length
        eventEmitter.emit('progress', downloadedB
      })
      .on('end', () => { // 7
        const data = Buffer.concat(chunks)
        cb(null, data)
      })
```

```
    })
    req.on('error', err => { // 8
      cb(err)
    })
    return eventEmitter // 9
  }
```

There’s a lot going on here, so let's walk through it step by step:

1. Our `download()` function takes a URL (`url`) to download from and a callback (`cb`) to handle the result once the download completes or if an error occurs.
2. It creates a new `EventEmitter` instance to emit events during the download process.
3. We use the Node’s core module `node:https` to make an HTTPS request, specifically we use the `get` function (`const req = get(url, resp => { ... })`). This function sends an HTTPS request to the provided URL and starts processing the response once the server responds. It returns a request object (`req`), an `EventEmitter` that we can use to keep track of the progress of forwarding the request to the server. The callback receives a response object (`resp`), which is another `EventEmitter` that can notify us of various events related to the response that we are receiving from the server. This includes events such as errors (`'error'`), new data received (`'data'`), and completion (`'end'`).

4. The response object is created once the server successfully sends the response headers, allowing us to access those headers. At this point, we initialize the `chunks` array to store the bytes that will be received over time (in chunks), along with the `downloadedBytes` counter to track the total number of bytes downloaded. By reading the `'Content-Length'` header, we can determine the total number of bytes expected in the response.
5. We add a listener for the `'error'` event on the response object to catch any issues while receiving the response body. If an error occurs, we pass it back to the caller through the callback.
6. Progress tracking is done by using the `'data'` event listener which is triggered each time a chunk of data is received. Each chunk is pushed into the `chunks` array, and `downloadedBytes` is updated with the size of the received chunk. The `eventEmitter` emits a `'progress'` event with the current number of downloaded bytes and the total file size, which can be used to calculate and display download progress.
7. When the response body has been fully sent, the `'end'` event fires. In our event handler, all chunks are concatenated into a single Buffer using `Buffer.concat(chunks)`, which represents the complete file data. The callback `cb(null, data)` is also called to signal completion and provide the downloaded data back to the caller.
8. If an error occurs during the request phase, the `req.on('error', err => { ... })` listener is triggered, and

the callback is called with the error. This allows us to capture and propagate errors that can happen during the request phase.

9. The function returns the `eventEmitter`, allowing the caller to listen for `'progress'` events to track download.

Yes, that was quite a bit of detail, but now let's see how to use the new `download()` function:

```
download(
  'https://example.com/somefile.zip',
  (err, data) => {
    if (err) {
      return console.error(`Download failed: ${er
    }
    console.log('Download completed', data)
  }
).on('progress', (downloaded, total) => {
  console.log(
    `${downloaded}/${total} ` +
    `(${((downloaded / total) * 100).toFixed(2)}%
  )
})
```

Here's how it works:

- Calling `download()`: You provide the URL of the file to download and a callback function. This callback handles the final result of the

download — either receiving the buffer with the downloaded data or an error.

- Progress Event Listener: You can also optionally attach a `'progress'` event listener. This listener prints the amount of data downloaded so far and the percentage of completion, providing nice progress feedback for the use in case the download can take some time.

An example output showing real-time progress updates and the final result could look like this:

```
280/127454 (0.22%)
1649/127454 (1.29%)
…
126747/127454 (99.45%)
127454/127454 (100.00%)
Download completed <Buffer ff d8 ff e0 00 10 4 a
```

Please note that this implementation is for illustrative purposes and not production-ready. It is simplified to focus on demonstrating the pattern of combining a callback with an `EventEmitter` *but lacks several important features. It doesn't handle chunked transfers, redirects, or plain HTTP requests, and it stores all data in a single buffer, which can lead to memory issues with large files. Error handling for HTTP response codes is also missing, and there's no support for download resumption or retries. For production scenarios, you might*

want to use a more comprehensive abstraction or library like `fetch` *or third-party options such as* `axios` *(https://nodejsdp.link/axios) offer advanced features and better error handling, making them more suitable for real-world applications.*

This example highlights the effectiveness of combining a callback with an `EventEmitter`. The key takeaway is that this pattern is especially useful for managing long-running tasks where you need to track both the completion and real-time progress. It's also applicable to various other practical scenarios, including file uploads, data processing pipelines, database backups and migrations, port scanning, brute-force applications, and more. Interestingly, the `get()` function from the `node:https` module leverages this pattern as well: it provides a callback for accessing the response while also returning an `EventEmitter` to monitor the ongoing request.

Although combining callbacks and event emitters is common in the Node.js ecosystem, this pattern is being gradually replaced by combining promises and async iterators. We will dive into these modern approaches in Chapter 9, Behavioral Design Patterns. Additionally, you can combine event emitters with other asynchronous mechanisms like promises. For example, you can return an object containing both a promise and an `EventEmitter`*, which can then be destructured by the caller, like this:* `{ promise, events } = foo()`.

Summary

In this chapter, we made our first contact with the practical aspects of writing asynchronous code. You discovered the two pillars of the entire Node.js asynchronous infrastructure—the callback and the `EventEmitter`—and we explored in detail their use cases, conventions, and patterns. We also explored some of the pitfalls of dealing with asynchronous code and you learned about the ways to avoid them. Mastering the content of this chapter paves the way toward learning the more advanced asynchronous techniques that will be presented throughout the rest of this book.In the next chapter, you will learn how to deal with complex asynchronous control flows using callbacks.

Exercises

- **3.1 A simple event**: Modify the asynchronous `FindRegex` class so that it emits an event when the find process starts, passing the input files list as an argument. Hint: beware of Zalgo!
- **3.2 Ticker**: Write a function that accepts a `number` and a `callback` as the arguments. The function will return an `EventEmitter` that emits an event called `tick` every 50 milliseconds until the `number` of milliseconds is passed from the invocation of the function. The function will also call the `callback` when the `number` of milliseconds has passed, providing, as the result, the total count of `tick` events emitted. Hint: you can use `setTimeout()` to schedule another `setTimeout()` recursively.

- **3.3 A simple modification**: Modify the function created in exercise 3.2 so that it emits a `tick` event immediately after the function is invoked.
- **3.4 Playing with errors**: Modify the function created in exercise 3.3 so that it produces an error if the timestamp at the moment of a `tick` (including the initial one that we added as part of exercise 3.3) is divisible by 5. Propagate the error using both the callback and the event emitter. Hint: use `Date.now()` to get the timestamp and the remainder (`%`) operator to check whether the timestamp is divisible by 5.
- **3.5 Disk bloat finder**: Create a function that accepts the path to a folder on the local file system and identifies the largest file within that folder. For extra credit, enhance the function to search recursively through subfolders. Hint: you can use the `node:fs` module for this task, specifically the `stats()` function to determine if a path is a directory or file and to get the file size in bytes, and the `readdir()` function to list the contents of a directory.

4

Asynchronous Control Flow Patterns with Callbacks

Transitioning from a synchronous programming style to a platform like Node.js, where **continuation-passing style (CPS)** and asynchronous APIs are the standard, can be challenging. Asynchronous code makes it difficult to predict the order in which statements will execute. Simple tasks such as iterating through a set of files, executing tasks sequentially, or waiting for multiple operations to complete require developers to adopt new approaches and techniques to avoid writing inefficient and hard-to-read code.When using callbacks to handle asynchronous control flow, the most common mistake is to fall into the trap of "callback hell," where code grows horizontally with excessive nesting, making even simple routines difficult to read and maintain. However, by applying discipline and utilizing certain patterns, it's possible to tame callbacks and write clean, manageable asynchronous code.In this chapter, we will deep dive on callbacks, what’s good about them, what’s bad, and how to handle them effectively. Specifically, we'll cover:

- The challenges of asynchronous programming.
- Best practices for avoiding callback hell and writing efficient asynchronous code.

- Common asynchronous patterns: Sequential execution: (executing tasks one after another), Sequential iteration (processing items in a sequence without parallelizing tasks), Concurrent execution (running multiple tasks concurrently), Limited concurrent execution (controlling the number of concurrent operations).

By mastering these concepts, you'll be well on your way to writing efficient and readable asynchronous code.

The challenges of asynchronous programming

Asynchronous programming in JavaScript may seem straightforward, but it's not without its pitfalls. The use of closures and in-place definitions of anonymous functions allows for a smooth coding experience that doesn't require the developer to jump to other points in the codebase – an approach aligned with the **KISS principle** (**Keep It Simple, Stupid** or, as we prefer, "**Keep It Super Simple**").

> *The KISS principle, is a friendly reminder to prioritize simplicity in software design. Coined by engineer Kelly Johnson at Lockheed, the idea was that things should be easy to fix, even with limited tools. For us developers, it means writing clear, understandable code over complex solutions. Aim for readability, use clear names, and stick to well-known patterns. This not only reduces bugs but also makes teamwork smoother and code easier to maintain.*

This flexibility makes it simple to write code that gets the job done quickly. However, this simplicity often comes at the cost of modularity, reusability, and maintainability. The risk is to end up with code that is made up of multiple nested callbacks, large functions and, overall, poor code organization.While there's no single "perfect" solution for dealing with the challenges of asynchronous code, we will equip you with the skills to evaluate tradeoffs and choose approaches that balance simplicity with maintainability. We'll learn how to recognize potential issues before they arise, anticipate problems, and implement solutions that promote modularity, reusability, and maintainability. By mastering these concepts, you'll be able to write efficient, readable, and well-structured asynchronous code that meets the demands of modern JavaScript development – not by eliminating all trade-offs but by understanding how to make informed decisions about when to prioritize simplicity over complexity.

Creating a simple web spider

To work on a practical and realistic example that can illustrate some of the challenges of asynchronous programming, we will create a little web spider: a command-line application that takes in a web URL as input and downloads its contents locally into one or more files. This kind of software is something that can be useful if you want to be able to browse a website offline or to take a snapshot of it at a certain point in time.I had to build something pretty similar once for work! I needed a program that could go through a website and find any links that were broken (404 errors). Using

asynchronous programming was the key to handle the scale of the sites it was used on, often composed by hundreds or even thousands of pages.These kinds of programs are really common when you're working with websites, and the web spider we are going to build will be a great starting point for your own projects. You can easily change how it works or add new features if you need to build something similar. For example, you might want to make a program that can crawl and extract structured information from multiple sites concurrently (i.e. a *web scraper*).In our implementation, we will often refer to a local module named `./utils.js`, which contains some helpers that we will be using in our application. We will omit the contents of this file for brevity, but you can find the full implementation, along with a `package.json` file containing the full list of dependencies, in the official repository at `nodejsdp.link/repo4`.Our application's core functionality is contained within the `spider.js` module. Let's take a look inside.First, we import the necessary dependencies:

```
import { writeFile } from 'node:fs'
import { dirname } from 'node:path'
import { exists, get, recursiveMkdir, urlToFilena
```

Here's a brief explanation of each utility function imported from `./utils.js`:

- `exists`: A callback-based function that checks if a file exists in the filesystem. It takes a file path and invokes the callback with a boolean

value indicating whether the file exists.

- `get`: A callback-based function that retrieves the body of an HTTP response for a given URL. It takes the URL as input and returns a `Buffer` representing the response body.
- `recursiveMkdir`: Another callback-based function that creates all necessary directories recursively within a specified path.
- `urlToFilename`: A synchronous function that converts a URL into a valid file system path.

Next, we define a new function named `spider()`, which takes two parameters: the URL to download and a callback function invoked when the download process completes:

```
export function spider(url, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => { // (
    if (err) { // (1.1)
      cb(err)
    } else if (alreadyExists) { // (1.2)
      cb(null, filename, false)
    } else { // (1.3)
      console.log(`Downloading ${url} into ${file
      get(url, (err, content) => { // (2)
        if (err) {
          cb(err)
        } else {
```

```
            recursiveMkdir(dirname(filename), err =
              if (err) {
                cb(err)
              } else {
                writeFile(filename, content, err =>
                  if (err) {
                    cb(err)
                  } else {
                    cb(null, filename, true) // (5)
                  }
                })
              }
            })
          }
        })
      }
    })
  }
```

There is a lot going on here, so let's discuss in more detail what happens in every step:

1. The code uses the `exists()` function to check whether the URL was already downloaded by verifying that the corresponding file is already present on disk. We can have 3 possible outcomes here.
   1. Outcome 1.1: we get an error (`err` is defined). This means that there was an error in accessing the filesystem. In this case we propagate the error back by invoking `cb(err)` .

    2. Outcome 1.2: The file already exists on the disk (`alreadyExists` is `true`). We don't want to download it again; we can simply call the callback with the `filename` and `false` (which indicates to the caller the file was not actively downloaded).
2. Outcome 1.3: The file does not exist (`alreadyExists` is `false`). We need to download the file from the URL.
3. To download the file we use the `get()` function.
4. We need to ensure that the destination folder exists. We can do that by using the `recursiveMkdir()` function.
5. Finally, we are ready to write the downloaded content to disk using Node.js `writeFile()` function.
6. If all went well, we can call the callback with the `filename` and `true` (this time indicating that the file was downloaded).

To use our web spider application, we can create a simple Command-Line Interface (CLI) application that reads a URL as an argument. This can be achieved by creating a new file called `spider-cli.js`:

```
import { spider } from './spider.js'
spider(process.argv[2], (err, filename, downloade
  if (err) {
    console.error(err)
    process.exit(1)
  }
  if (downloaded) {
    console.log(`Completed the download of "${fi
```

```
    } else {
      console.log(`"${filename}" was already downlo
    }
  })
```

If you've copied the `utils.js` and `package.json` files from our code samples repository, run `npm install` to download all necessary dependencies. You're now ready to test your web spider application.To run it, navigate to your project directory in your terminal or command prompt and execute:

```
node spider-cli.js https://www.nodejsdesignpatter
```

This should create a file called `www.nodejsdesignpatterns.com/index.html` starting from the current working directory. Feel free to check the content of this file, compare it with the source HTML code of the remote website, and to play around with other URLs.

> *Note that our simple web spider application requires that we always include the protocol (for example, `http://`) in the URL we provide. Also, do not expect HTML links to be rewritten or resources such as images, videos, scripts and stylesheets to be downloaded. We are intentionally keeping this example lean, because we want to focus on demonstrating how asynchronous programming works. Feel free to try adding these extra features yourself as a fun challenge!*

In the next section, you will learn how to improve the readability of this code and, in general, how to keep callback-based code as clean and readable as possible.

Callback hell

Take a closer look at the `spider()` function. You'll see that even with a simple algorithm, the code becomes deeply nested and hard to follow. This is a common issue with asynchronous code.Now, imagine if we could use a blocking API instead. The code would be much simpler and easier to read. To give you a better idea, here's what the implementation could look assuming we had equivalent blocking APIs for all our helper functions:

```
export function spider(url) {
  const filename = urlToFilename(url)
  if (exists(filename)) {
    return false
  } else {
    const content = get(url)
    recursiveMkdir(dirname(filename))
    writeFile(filename, content)
  }
}
```

Notice how we don't even need to handle errors explicitly; any synchronous exception will automatically propagate up the call stack.However, things

are different when it comes to using asynchronous CPS. Defining callbacks in place can quickly lead to messy and unreadable code.This problem, where an overload of closures and inline callbacks turns the code into an unmanageable tangle, is commonly known as **callback hell**. It's one of the most notorious anti-patterns in Node.js and JavaScript development. Code caught in this trap typically looks something like this:

```
asyncFoo(err => {
  asyncBar(err => {
    asyncFooBar(err => {
      //...
    })
  })
})
```

You can see how code written in this way assumes the shape of a pyramid (pointing to the right) due to deep nesting, and that's why it is also colloquially known as the **pyramid of doom**.The most obvious problem with code like this is its poor readability. The deep levels of nesting can make it really hard to tell where one function ends and another begins.Another issue arises from overlapping variable names across different scopes. Often, we end up using similar, or even identical, names for variables in each scope. A common example is the error argument passed to each callback. Some developers try to differentiate these by using slight variations, like `err`, `error`, `err1`, `err2`, and so on. Others prefer to use the same name, such as

`err`, across all scopes, shadowing the variable from the outer scope. Neither of these approaches is ideal, as both increase confusion and the risk of introducing bugs.Additionally, closures come with some performance and memory overhead. They can also cause memory leaks, which aren't always easy to track down. In fact, any context referenced by an active closure is retained and won't be cleaned up by garbage collection.

> *Remember that, when a closure is created, it "closes over" its surrounding lexical environment, meaning it retains access to variables and data in that environment, even after the outer function has completed. This is the core of how closures work, but it also means that the referenced context cannot be garbage collected as long as the closure is still in use. In essence, any variable, object, or other data within the closure's scope will remain in memory.*

If you revisit our `spider()` function, you'll see it's a prime example of callback hell, displaying all the issues we just discussed. Fortunately, we'll tackle these problems using the patterns and techniques covered in the upcoming sections of this chapter.

Callback best practices

Now that you've encountered your first example of callback hell, you know what to avoid. But managing asynchronous code comes with more challenges than just preventing deeply nested callbacks. In fact, there are many

situations where controlling the flow of multiple asynchronous tasks requires specific patterns and techniques, especially if you're working with plain JavaScript without any external libraries.For example, consider the common scenario where you have an array of URLs and you need to fetch the content of each URL **in sequence**, one after the other. In real life, one reason why you might want to do something like this sequentially could be due to strict rate-limiting restrictions (and prevent doing more than one request at the time) or other sequential dependencies. To solve this problem, you might be tempted to use a simple `forEach()` loop, like the following:

```
const urls = ['url1', 'url2', 'url3']
urls.forEach(url => {
  fetch(url, response => {
    console.log(`Fetched ${url}`)
    // ... process the response
  })
})
```

However, this code won't work as you might expect. The `forEach()` loop will execute synchronously, initiating all the `fetch` calls very quickly, one after the other. Because `fetch` is asynchronous, the callbacks will be executed at an unspecified point in the future without regard to the order in the array. This would lead to requests being made concurrently and to results that are not in order, with no clear control on execution.To process

these asynchronous operations sequentially, we need a mechanism that awaits the completion of each operation before moving on to the next. A common technique for this type of problem is to implement a pattern similar to recursion, which is something we will explore later in this chapter.In this section, you'll not only learn how to steer clear of callback hell, but also how to implement some of the most common control flow patterns — using nothing but simple, plain JavaScript.Implementing various control flows using callbacks can become cumbersome quickly if we are not careful. Before we dive into the details of how we can use callbacks to implement different control flows, let's take a short break to explore a few techniques that will help us write higher quality, more maintainable, callback-based code. Applying these techniques is key to making working with callbacks more manageable and enjoyable.

Callback discipline

When writing asynchronous code, the first rule is to avoid overusing in-place function definitions for callbacks. While it might seem convenient, as it saves you from worrying about modularity or reusability, you've seen how it often leads to more problems than it solves. In most cases, fixing callback hell doesn’t require libraries, complex techniques, or a shift in paradigms — it just takes following some simple ideas.Here are a few basic principles that can help reduce nesting and improve code organization overall:

- **Exit early**: Use `return`, `continue`, or `break` as needed to exit a statement immediately, instead of nesting full `if...else` blocks. This keeps your code shallow and easier to follow.
- **Use named functions for callbacks**: Move callbacks out of closures, and pass intermediate results as arguments. Named functions also provide clearer stack traces, which is helpful for debugging.
- **Modularize your code**: Break your code into smaller, reusable functions whenever possible to promote clarity and maintainability.

Now, let's apply these principles in practice.

Applying the callback discipline

To illustrate the effectiveness of the callback principles we mentioned, let’s apply them to resolve the callback hell in our web spider application.First, we can refactor our error-checking pattern by removing the `else` statement. This works because we can return from the function immediately when an error is detected. Instead of writing:

```
if (err) {
  cb(err)
} else {
  // code to execute when there are no errors
}
```

We can streamline the code like this:

```
if (err) {
  return cb(err)
}
// code to execute when there are no errors
```

This is often referred to as the **early return principle**. With this simple trick, we immediately have a reduction in the nesting level of our functions. It is easy and doesn't require any complex refactoring.

> *A common mistake when executing the optimization just described is forgetting to terminate the function after the callback is invoked. For an error-handling scenario, the following code is a typical source of defects:*
>
> ```
> if (err) {
> cb(err)
> }
> // code to execute when there are no errors.
> ```
>
> *We should never forget that the execution of our function will continue even after we invoke the callback. It is then important*

to insert a `return` *instruction to block the execution of the rest of the function. Also, note that it doesn't really matter what value is returned by the function; the real result (or error) is produced asynchronously and passed to the callback. The return value of the asynchronous function is usually ignored. This property allows us to write shortcuts such as the following:*

```
return cb(...)
```

Otherwise, we'd have to write slightly more verbose code, such as the following:

```
cb(...)

return
```

As a second optimization for our `spider()` function, we can try to identify reusable pieces of code. For example, the functionality that writes a given string to a file (also making sure that the directory path is created if needed) can be easily factored out into a separate function, as follows:

```
function saveFile(filename, content, cb) {
  recursiveMkdir(dirname(filename), err => {
    if (err) {
      return cb(err)
    }
    writeFile(filename, content, cb)
```

```
    })
  }
```

Following the same principle, we can create a generic function named `download()` that takes a URL and a filename as input, and downloads the URL into the given file. Internally, we can use the `saveFile()` function we created earlier:

```
function download(url, filename, cb) {
  console.log(`Downloading ${url} into ${filename
  get(url, (err, content) => {
    if (err) {
      return cb(err)
    }
    saveFile(filename, content, err => {
      if (err) {
        return cb(err)
      }
      cb(null, content)
    })
  })
}
```

Now, our code is also more modular and explicit. To finish the refactoring, we modify the `spider()` function, which will now look like the following:

```
export function spider(url, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      return cb(err)
    }
    if (alreadyExists) {
      return cb(null, filename, false)
    }
    download(url, filename, err => {
      if (err) {
        return cb(err)
      }
      cb(null, filename, true)
    })
  })
}
```

The functionality and interface of the `spider()` function remain unchanged; what we've improved is how the code is organized. By applying the early return principle and other callback discipline techniques, we significantly reduced code nesting while increasing both reusability and testability. In fact, we could consider exporting both `saveFile()` and `download()` and moving them in our shared utility library to make them reusable across different modules. This would also allow us to test these functions in isolation, improving our code quality.This refactoring shows

that, more often than not, all it takes is some discipline to avoid overusing closures and anonymous functions. It's a simple, effective approach that requires minimal effort and doesn't rely on external libraries.Now that you know how to write clean, asynchronous code with callbacks, we're ready to dive into common asynchronous patterns, such as sequential and concurrent execution.

Control flow patterns

In this section, we will look at asynchronous control flow patterns and start by analyzing the sequential execution flow.

Sequential execution

Executing a set of tasks in sequence means running them one at a time, one after the other. The order of execution matters and must be preserved, because the result of a task in the list may affect the execution of the next. *Figure 4.1* illustrates this concept:

Figure 4.1: An example of sequential execution flow with three tasks

There are different variations of this flow:

- Executing a set of known tasks in sequence, without propagating data across them.
- Using the output of a task as the input for the next (also known as *chain*, *pipeline*, or *waterfall*).
- Iterating over a collection while running an asynchronous task on each element, one after the other.

Sequential execution is usually the main cause of the callback hell problem when using asynchronous CPS.

Executing a known set of tasks in sequence

You may not have realized it, but in the previous section, we already delved into the concept of sequential execution flow while implementing the `spider()` function. If you take a moment to think about it, you'll see that our spider performs several asynchronous tasks in a specific order, with each task completing before the next begins: checking if the file already exists, downloading content from a remote URL, and saving that content to a file. By following some simple rules, we were able to organize these tasks into a sequential flow. Now, using that code as a reference, we can generalize our solution with the following pattern:

```
function task1(cb) {
  asyncOperation(() => {
```

```
    task2(cb) // calls task2 with the current cal
  })
}
function task2(cb) {
  asyncOperation(() => {
    task3(cb) // calls task3 with the current cal
  })
}
function task3(cb) {
  asyncOperation(() => {
    cb() // finally completes and executes the ca
  })
}
task1(() => {
  // executed when task1, task2 and task3 are com
  console.log('tasks 1, 2 and 3 executed')
})
```

The pattern above demonstrates how each task triggers the next one upon the completion of a generic asynchronous operation. It highlights the modularization of tasks and shows that closures aren't always necessary for handling asynchronous code.

Sequential iteration

The sequential execution pattern works great when we know in advance which tasks need to be executed and how many there are, especially if the number of tasks is relatively small. This lets us hardcode the invocation of

each subsequent task in the sequence. But what happens when we want to perform an asynchronous operation for every item in a collection? In situations like this, we can't hardcode the task sequence anymore; instead, we need to build it dynamically.

Web spider version 2

To illustrate sequential iteration, let's introduce a new feature to our web spider application: the ability to download all links on a web page recursively, as long as the links stay within the same domain. After all, it's a spider—it can crawl through the web, following links into its depths! To accomplish this, we'll extract all the links from the current page and have our web spider follow each one, recursively downloading them in sequence.The first step is to modify our `spider()` function to trigger a recursive download of the page's links by using a new function called `spiderLinks()`, which we'll create shortly.A key design decision here is ensuring the spider doesn't get stuck in an endless loop. To prevent this, we'll introduce a `maxDepth` parameter that limits the recursion depth. Here's the updated code for the `spider()` function:

```
export function spider(url, maxDepth, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      // error checking the file
      return cb(err)
```

```
  }
  if (alreadyExists) {
    if (!filename.endsWith('.html')) {
      // ignoring non-HTML resources
      return cb()
    }
    return readFile(filename, 'utf8', (err, fi
      if (err) {
        // error reading the file
        return cb(err)
      }
      return spiderLinks(url, fileContent, maxD
    })
  }
  // The file does not exist, download it
  download(url, filename, (err, fileContent) =>
    if (err) {
      // error downloading the file
      return cb(err)
    }
    // if the file is an HTML file, spider it
    if (filename.endsWith('.html')) {
      return spiderLinks(url, fileContent.toStr
    }
    // otherwise, stop here
    return cb()
  })
```

```
    })
  }
```

We've added comments throughout the code, so it should be easy to follow what this new version does. One thing to note is that our spider can also download non-HTML resources, so we added checks to ensure it only crawls when the current URL points to a web page (by checking whether the file extension is `".html"`).

> *To keep things simple, we use some basic heuristics to determine if the URL is an HTML resource (you can find these in the `urlToFilename()` function in the `utils.js` file on our code repo). These checks are generally reliable but not bulletproof, so there may be situations where this approach doesn't behave as expected. A more robust solution would be to rely less on URL structure and instead use the content type provided by most web servers through the `Content-Type` header. If you're interested in learning more about the Content-Type header, check out `nodejsdp.link/content-type`.*

In the next section, we'll explore how the `spiderLinks()` function is implemented.

Sequential crawling of links

Now, we can build the core of this new version of our web spider: the `spiderLinks()` function. This function downloads all the links on an HTML page using a sequential asynchronous iteration algorithm. Take a close look at how we define it in the following code block:

```
function spiderLinks(currentUrl, body, maxDepth,
  if (maxDepth === 0) { // (1)
    // Remember Zalgo from Chapter 3?
    // To prevent that, this function is designed
    // invoke its callback asynchronously.
    return process.nextTick(cb)
  }
  const links = getPageLinks(currentUrl, body) //
  if (links.length === 0) {
    return process.nextTick(cb)
  }
  function iterate(index) { // (3)
    if (index === links.length) {
      return cb()
    }
    spider(links[index], maxDepth - 1, err => { /
      if (err) {
        return cb(err)
      }
      iterate(index + 1)
    })
  }
```

```
    iterate(0) // (5)
  }
```

Here are the key steps to understand in this new function:

1. Base case: If the `maxDepth` variable has reached 0, we stop crawling. Notice how we call the callback asynchronously to avoid the infamous Zalgo.
2. We extract all the links on the page using the `getPageLinks()` function, which returns only links pointing to the same hostname (this function is implemented in the `utils.js` file). If there are no links, we stop the process.
3. We use a local function called `iterate()` to go over the links. This function takes the index of the next link to process. The first thing it does is check if the index equals the length of the links array. If it does, it means we've processed all items, and we can stop by invoking `cb()`.
4. At this stage, everything is ready for processing the link. We call the `spider()` function, reducing the recursion depth (`maxDepth - 1`), and continue with the next step of the iteration once the operation completes.
5. As the final step in `spiderLinks()`, we start the iteration by calling `iterate(0)`.

The algorithm we just introduced allows us to iterate over an array by executing an asynchronous operation sequentially, which, in our case, is the `spider()` function.Now, let's update `spider-cli.js` to allow us to

specify the nesting level as an additional command-line interface (CLI) argument:

```
import { spider } from './spider.js'
const url = process.argv[2]
const maxDepth = Number.parseInt(process.argv[3],
spider(url, maxDepth, err => {
  if (err) {
    console.error(err)
    process.exit(1)
  }
  console.log('Downloaded complete')
})
```

We can now test this new version of our spider application and watch as it downloads all the links on a web page recursively, one after the other. To stop the process—since it can take a while if there are many links—remember that you can always press *Ctrl + C*. If you want to resume later, you can restart the spider with the same URL you used in the first run.

> *Now that our web spider has the potential to download an entire website, please use it with care. For example, avoid setting a high max depth level or leaving the spider running for too long. Overloading a server with thousands of requests is not only impolite, but in some cases, it may even be illegal. For example, it could be considered a DoS (Denial of Service) attack. Spider responsibly!*

The pattern

The code of the `spiderLinks()` function from the previous section is a clear example of how it's possible to iterate over a collection while applying an asynchronous operation. You may also notice that it's a pattern that can be adapted to any other situation where we need to iterate asynchronously over the elements of a collection or, in general, over a list of tasks. This pattern can be generalized as follows:

```
function iterate (index) {
  if (index === tasks.length) {
    return finish()
  }
  const task = tasks[index]
  task(() => iterate(index + 1))
}
function finish () {
  // iteration completed
}
iterate(0)
```

It's important to notice that these types of algorithms become really recursive if `task()` is a synchronous operation. In such a case, the stack will not unwind at every cycle and there might be a risk of hitting the maximum call stack size limit.

The pattern that was just presented is very powerful and can be extended or adapted to address several common needs. Just to mention some examples:

- We can map the values of an array asynchronously.
- We can pass the results of an operation to the next one in the iteration to implement an asynchronous version of the `reduce` algorithm.
- We can quit the loop prematurely if a particular condition is met (asynchronous implementation of the `Array.some()` helper).
- We can even iterate over an infinite number of elements.

We could also choose to generalize the solution even further by wrapping it in a function with a signature such as the following:

```
iterateSeries(collection, iteratorCallback, final
```

Here, `collection` is the actual dataset you want to iterate over, `iteratorCallback` is the function to execute over every item, and `finalCallback` is the function that gets executed when all the items are processed or in case of an error. The implementation of this helper function is left to you as an exercise.

The Sequential Iterator pattern

> *Enables the execution of tasks within a collection, one after the other, ensuring a controlled flow. This is achieved by using an*

`iterator` *function that automatically triggers the next task in the sequence as soon as the current one completes.*

In the next section, we will explore the concurrent execution pattern, which is more convenient when the order of the various tasks is not important.

Concurrent execution

There are some situations where the order of execution of a set of asynchronous tasks is not important and there is no logical correlation or any data dependency between these tasks. All we want is to be notified when all those running tasks are completed. Such situations are better handled using a concurrent execution flow.At this point in the book, we have used the words **parallel** and **concurrent** a few times. They might sound similar, but the difference between them is an important concept to understand, especially in the context of Node.js.As we discussed in *Chapter 1*, *The Node.js Platform*, Node.js is single-threaded, which makes understanding the concept of concurrency particularly relevant. Before we continue, let's try to define these two concepts with an analogy.Imagine you run a restaurant and receive two orders at the same time. In the kitchen, there are two ways to handle this:

- **Parallelism:** If you have two chefs, each can prepare an order simultaneously, working completely independently. This is like having multiple CPU cores or worker threads executing tasks in parallel.

- **Concurrency**: If there's only one chef, they need to divide their time efficiently. While waiting for water to boil for one dish, they can start chopping ingredients for the other. The chef isn't working on both tasks simultaneously but is making progress on both. This is how an event loop works, switching between tasks efficiently while waiting on slower operations like I/O.

Node.js is built on JavaScript, which runs on a single-threaded event loop. This means that, by default, Node.js excels at concurrent execution, handling multiple tasks without blocking the main thread. While parallelism can be achieved using worker threads or by spawning multiple processes, concurrency is often more efficient and lightweight for many real-world applications, such as handling multiple network requests or database queries.A parallel execution of multiple tasks might be represented as in *Figure 4.2*:

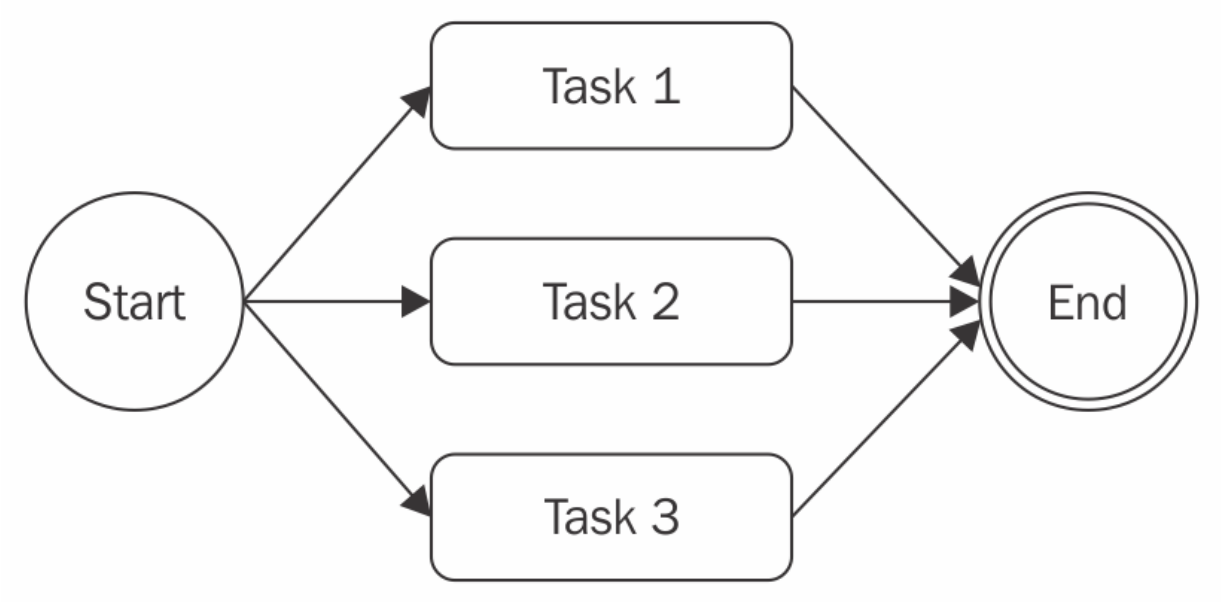

Figure 4.2: An example of parallel execution with three tasks

Although this picture is generally used to represent parallel execution of multiple tasks, it can also help describe concurrency at a high level, where multiple tasks are actively progressing within the event loop. The key distinction between parallel and concurrent execution lies in how progress is made: parallel execution runs tasks simultaneously, while concurrency involves efficiently switching between tasks to keep things moving.So to further clarify our understanding, let's look at another *diagram that shows how two asynchronous tasks can run concurrently in a Node.js program*:

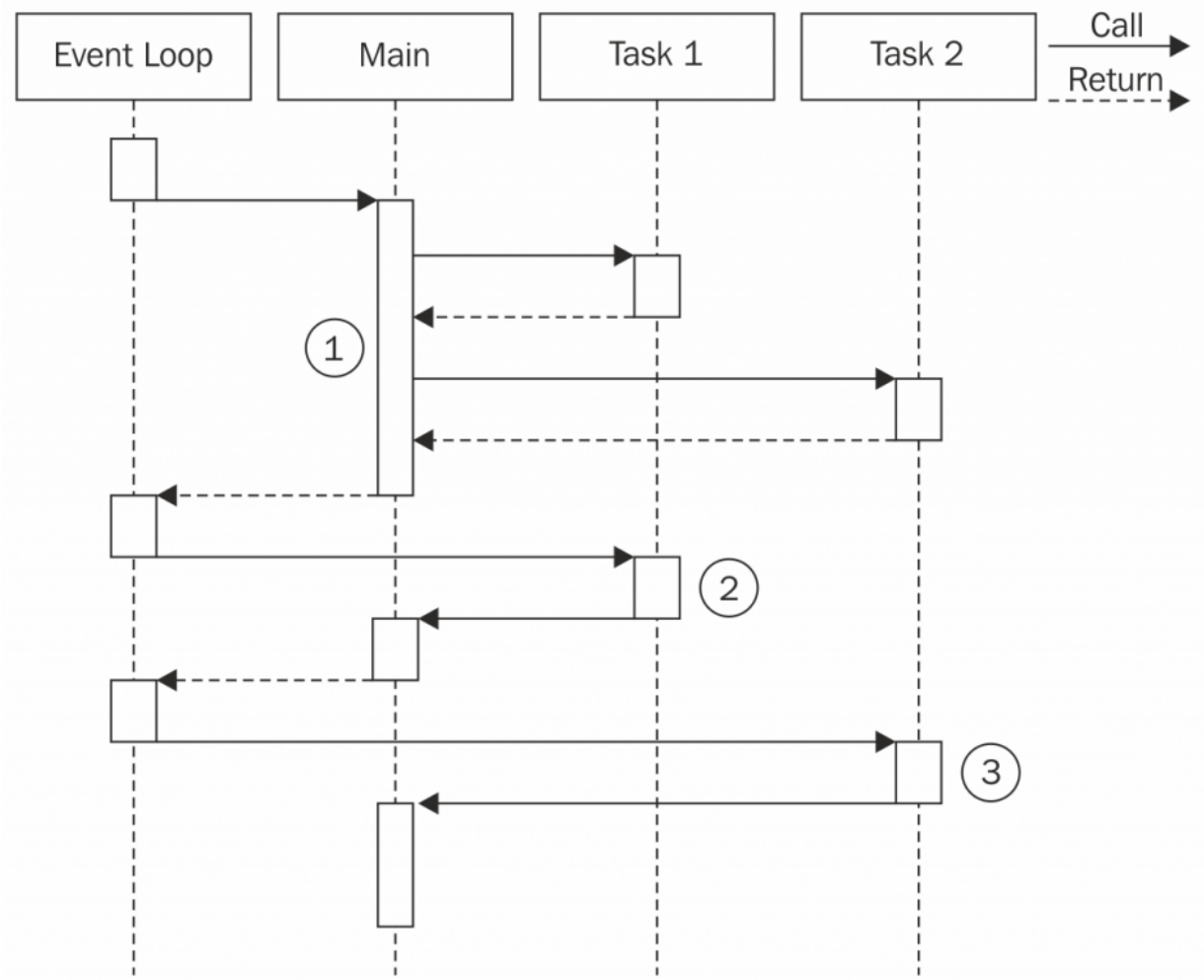

Figure 4.3: An example of how asynchronous tasks run concurrently

In *Figure 4.3*, we have a **Main** function that executes two asynchronous tasks:

1. The **Main** function triggers the execution of **Task 1** and **Task 2**. As they trigger an asynchronous operation, they immediately return control back to the **Main** function, which then returns it to the event loop.

2. When the asynchronous operation of **Task 1** is completed, the event loop gives control to it. When **Task 1** completes its internal synchronous processing as well, it notifies the **Main** function.
3. When the asynchronous operation triggered by **Task 2** is complete, the event loop invokes its callback, giving control back to **Task 2**. At the end of **Task 2**, the **Main** function is notified once more. At this point, the **Main** function knows that both **Task 1** and **Task 2** are complete, so it can continue its execution or return the results of the operations to another callback.

In short, this means that in Node.js, we generally execute asynchronous operations concurrently, because their concurrency is handled internally by the non-blocking APIs. In Node.js, synchronous (blocking) operations can't be easily parallelised or run concurrently unless their execution is interleaved with an asynchronous operation, or interleaved with `setTimeout()` or `setImmediate()`. You will see these techniques in more detail in *Chapter 11*, *Advanced Recipes*.

Web spider version 3

Our web spider application seems like a perfect candidate to apply the concept of concurrent execution. So far, our application is executing the recursive download of the linked pages in a sequential fashion. We can easily improve the performance of this process by downloading all the linked pages concurrently.To do that, we just need to modify the `spiderLinks()`

function to make sure we spawn all the `spider()` tasks at once, and then invoke the final callback only when all of them have completed their execution. So, let's modify our `spiderLinks()` function as follows:

```
function spiderLinks(currentUrl, body, maxDepth,
  if (maxDepth === 0) {
    return process.nextTick(cb)
  }
  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return process.nextTick(cb)
  }
  let completed = 0
  let hasErrors = false
  function done(err) { // (2)
    if (err) {
      hasErrors = true
      return cb(err)
    }
    if (++completed === links.length && !hasErro
      return cb()
    }
  }
  for (const link of links) { // (1)
    spider(link, maxDepth - 1, done)
  }
}
```

Let's discuss what we changed.

1. As mentioned earlier, the `spider()` tasks are now started all at once. This is possible by simply iterating over the `links` array and starting each task without waiting for the previous one to finish
2. Then, the trick to make our application wait for all the tasks to complete is to provide the `spider()` function with a special callback, which we call `done()`. The `done()` function increases a counter when a `spider` task completes. When the number of completed downloads reaches the size of the `links` array, the final callback (`cb`) is invoked.

The `hasErrors` variable is necessary because if one concurrent task fails, we want to immediately call the callback with the given error. Also, we need to make sure that other concurrent tasks that might still be running won't invoke the callback again.With these changes in place, if we now try to run our spider against a web page, we will notice a huge improvement in the speed of the overall process, as every download will be carried out concurrently, without waiting for the previous link to be processed.

The pattern

Finally, we can extract our nice little pattern for the concurrent execution flow. Let's represent a generic version of the pattern with the following code:

```
const tasks = [ /* ... */ ]
let completed = 0
for (const task of tasks) {
  task(() => {
    if (++completed === tasks.length) {
      finish()
    }
  })
}
function finish () {
  // all the tasks completed
}
```

With small modifications, we can adapt the pattern to accumulate the results of each task into a collection, to filter or map the elements of an array, or to invoke the `finish()` callback as soon as one or a given number of tasks complete (this last situation in particular is called **competitive race**).

The Unlimited Concurrent Execution pattern

> *This pattern involves running a set of asynchronous tasks concurrently by launching them all at once and waiting for their completion. All tasks are started immediately, and completion is tracked by counting how many times their callbacks are invoked.*

When we have multiple tasks running concurrently, we might have race conditions, that is, contention to access external resources (for example, files or records in a database). In the next section, we will talk about race conditions in Node.js and explore some techniques to identify and address them.

Fixing race conditions with concurrent tasks

Running a set of tasks in parallel can cause issues when using blocking I/O in combination with multiple threads. However, you have just seen that, in Node.js, this is a totally different story. Running multiple asynchronous tasks concurrently is, in fact, straightforward and cheap in terms of resources.This is one of the most important strengths of Node.js, because it makes running multiple tasks through concurrency a common practice rather than a complex technique to only use when strictly necessary.Another important characteristic of Node.js's concurrency model is how it handles task synchronization and race conditions. In a multithreaded environment, managing shared resources typically requires synchronization mechanisms such as locks, mutexes, semaphores, and monitors. These constructs help coordinate access to shared data but can introduce significant complexity and performance overhead.To bring back our kitchen analogy, imagine two chefs working in parallel, each preparing their own dish. If they both need to use the same sink at the same time to wash ingredients, they can't proceed simultaneously; they must either take turns or risk getting in each other's way. In multithreaded programming, mechanisms like locks act as a

way to manage this contention, ensuring that only one task accesses the shared resource at a time. However, poorly managed synchronization can lead to inefficiencies, such as one chef blocking the sink for too long, slowing down the entire kitchen.In Node.js, we usually don't need a fancy synchronization mechanism, as everything runs on a single thread. However, this doesn't mean that we can't have race conditions; on the contrary, they can be quite common. The root of the problem is the delay between the invocation of an asynchronous operation and the notification of its result.To see a concrete example, we will refer again to our web spider application, and in particular, the last version we created, which contains a race condition. Can you spot it? Take a few minutes to think about it and see if you can guess what's the issue. We'll be here waiting for you!The problem we are talking about lies in the `spider()` function, where we check whether a file already exists before we start to download the corresponding URL:

```
export function spider(url, maxDepth, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    // ...
    if (alreadyExists) {
      // ...
    } else {
      download(url, filename, (err, fileContent)
        // ...
```

The problem is that two `spider` tasks operating on the same URL might invoke `readFile()` on the same file before one of the two tasks completes the download and creates a file, causing both tasks to start a download. *Figure 4.4* explains this situation:

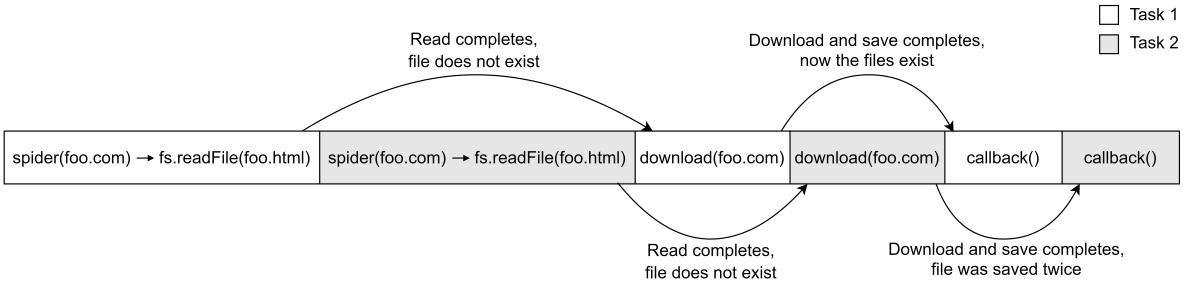

Figure 4.4: An example of a race condition in our spider() function

Figure 4.4 shows how **Task 1** and **Task 2** are interleaved in the single thread of Node.js, as well as how an asynchronous operation can actually introduce a race condition. In our case, the two `spider` tasks end up downloading the same file.How can we fix that? The answer is much simpler than you might think. In fact, all we need is a variable to mutually exclude multiple `spider()` tasks running on the same URL. This can be achieved with some code, such as the following:

```
const spidering = new Set()
function spider (url, nesting, cb) {
  if (spidering.has(url)) {
    return process.nextTick(cb)
  }
```

```
    spidering.add(url)
    // ...
```

We simply exit the function right away if the URL is already in the global `spidering` set (I know *spidering* isn't technically a word, but it sounds cool, doesn't it?). Otherwise, we add the URL to the set and proceed with the download. Since we don't want to download a URL more than once, we don't need to remove URLs from the set.

> *If you are building a spider that might have to download hundreds of thousands of web pages, removing the downloaded* `url` *from the set once a file is downloaded will help you to keep the set cardinality, and therefore the memory consumption, from growing indefinitely.*

Race conditions can cause all sorts of issues, even in a single-threaded environment like Node.js. In some cases, they can lead to data corruption and are notoriously difficult to debug due to their fleeting nature. That's why it's always a good idea to double-check for potential race conditions when running tasks concurrently.Speaking of concurrent tasks, running too many concurrent tasks at once is usually a bad idea—you could quickly exhaust system memory or run into limits like the maximum number of open file descriptors. In the next section, we'll dive into why this can be a problem and how to manage the number of concurrent tasks effectively.

Limited concurrent execution

Spawning concurrent tasks without any control can easily lead to excessive load. Imagine trying to read thousands of files, access multiple URLs, or execute numerous database queries all at once. The most common issue in such cases is running out of resources. For example, an application might attempt to open too many files at the same time, quickly exhausting the available file descriptors.

> *A server that launches an unlimited number of concurrent tasks in response to user requests can also become vulnerable to a* **denial-of-service** *(***DoS***) attack. In this type of attack, a malicious actor can craft one or more requests to the server that somehow force it to use up all its resources and become unresponsive. Limiting the number of concurrent tasks is a good practice that helps build more resilient applications.*

Currently, Version 3 of our web spider doesn't impose any limits on concurrent tasks, making it prone to crashing in several scenarios. For instance, if we run it against a large website, it might run for a few seconds before failing with an `ECONNREFUSED` error. This happens because, when too many pages are being downloaded concurrently, the web server might start rejecting new connections from the same IP. In such cases, our spider would crash, and we'd have to restart the process to continue crawling the site. While we could handle the `ECONNREFUSED` error to prevent the crash, we'd still risk overloading the system by allocating too many concurrent tasks, leading to other problems.In this section, we'll explore how to make

our spider more resilient by limiting concurrency.The following diagram shows a scenario where we have 5 tasks to run concurrently, but we've set a concurrency limit of 2:

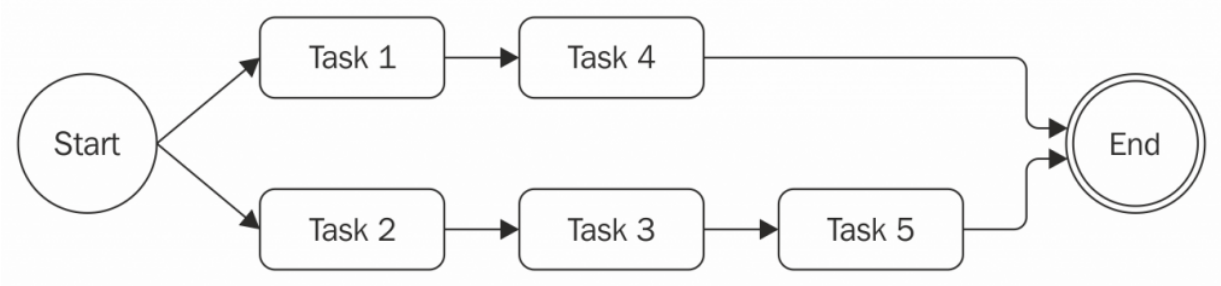

Figure 4.5: An example of how concurrency can be limited to a maximum of two tasks at the time

Looking at *Figure 4.5*, we can see that, with this particular example, the algorithm works like this:

- First, we start as many tasks as possible without going over the concurrency limit. Since the limit is set to two, the Start node launches Task 1 and Task 2.
- As soon as a task finishes, we launch the next one in line, always keeping the limit in mind. For example, when Task 2 finishes, Task 3 is started. When Task 1 finishes, Task 4 is started. Once Task 3 finishes, we move on to the final task, Task 5. When Task 5 completes, and there are no more tasks to run, the entire process is done.

In the next section, we will explore a possible implementation of the limited concurrent execution pattern.

Limiting concurrency

We will now look at a pattern that will execute a set of given tasks concurrently with limited concurrency:

```
const tasks = [
  // ...
]
const concurrency = 2
let running = 0
let completed = 0
let nextTaskIndex = 0
function next() { // (1)
  while (running < concurrency && nextTaskIndex <
    const task = tasks[nextTaskIndex++]
    task(() => { // (2)
      if (++completed === tasks.length) {
        return finish()
      }
      running--
      next()
    })
    running++
  }
}
```

```
next()
function finish() {
  // all the tasks completed
}
```

This algorithm can be considered a mixture of sequential execution and concurrent execution. In fact, you might notice similarities with both patterns:

1. We have a function called `next()` that acts as an iterator. Inside it, there's a loop that starts as many tasks as possible, while keeping the number of running tasks within the set concurrency limit.
2. Each time a task is started, we pass it a callback. In that callback, we decide what to do when the task completes. If there are more tasks left, we call `next()` again to keep things going. If all tasks are finished, we call `finish()` to signal that everything is done. Along the way, we make sure to properly update our counters: increasing the count of completed tasks and adjusting the number of running tasks (which gets increased each time `next()` starts a new task).

Pretty simple, isn't it?

> *In this pattern, we control concurrency by tracking how many tasks are running at any given time using the `running` variable. Since Node.js runs JavaScript in a single-threaded event loop, there's no risk of multiple tasks modifying `running` at the same time; each*

update happens in sequence, making synchronization unnecessary. As tasks start, `running` *increases, and as they complete, it decreases before triggering* `next()` *to launch more tasks if needed. Because execution is always ordered within the event loop,* `running` *is always accurate, without the complexity of locks or race conditions. This is a key advantage of Node.js's concurrency model: efficient task management without the pitfalls of multithreading.*

Globally limiting concurrency

Our web spider application is a great example of where we can apply what we've just learned about limiting the number of tasks running at once. Without such a limit, we might end up crawling thousands of links simultaneously, which could overwhelm our system. By controlling the number of concurrent downloads, we can add some much-needed predictability to the process.Now, we could apply the limited concurrency pattern to our `spiderLinks()` function. However, this would only limit the number of tasks for the links found on a single page. For instance, if we set the concurrency limit to two, we'd have at most two links downloading at the same time *per page*. But since each page might spawn two more downloads, this could still cause the total number of downloads to grow quickly, potentially out of control.In general, this particular implementation of the limited concurrency pattern works well when you have a predefined set of tasks, or when tasks increase at a manageable rate. However, in cases like our web spider—where a task can spawn multiple new tasks—it's not effective for

limiting overall concurrency. To fix this, we need to introduce a mechanism that allows us to control concurrency on a global level.

Queues to the rescue

What we really want is to limit the total number of download operations running concurrently. A good way to achieve this is by introducing queues to manage the concurrency of multiple tasks. Let's see how this works.We'll now implement a simple class called `TaskQueue`, which combines a queue with the limited concurrency algorithm we just discussed. Let's create a new module called `taskQueue.js`:

```
export class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency
    this.running = 0
    this.queue = []
  }
  pushTask(task) {
    this.queue.push(task)
    process.nextTick(this.next.bind(this))
    return this
  }
  next() {
    while (
      this.running < this.concurrency &&
      this.queue.length > 0
```

```
      ) {
        const task = this.queue.shift()
        task(() => {
          this.running--
          process.nextTick(this.next.bind(this))
        })
        this.running++
      }
    }
  }
```

The constructor of this class takes, as input, only the concurrency limit, but besides that, it initializes the instance variables `running` and `queue`. The former variable is a counter used to keep track of all the running tasks, while the latter is the array that will be used as a queue to store the pending tasks.The `pushTask()` method simply adds a new task to the queue and then bootstraps the execution of the worker by asynchronously invoking `this.next()`.

> *In this example, `bind(this)` ensures that the `next()` method retains the correct context when executed asynchronously via `process.nextTick()`. When we pass `this.next()` directly to `process.nextTick()`, we're only passing a reference to the method, not invoking it immediately. This becomes an issue because, in JavaScript, when a method is called without an explicit object (like `queue.next()` – where queue is the context), it loses its original context. Since `next()` relies on `this.concurrency`,*

`this.running`, and `this.queue`, calling it in the wrong context means it won't have access to these properties, leading to unexpected behavior or errors. Using `this.next.bind(this)` ensures that `next()` is always executed with the correct `this` reference, preserving access to the instance's properties and keeping the task queue functioning as expected.

The `next()` method spawns a set of tasks from the queue, ensuring that it does not exceed the concurrency limit.You may notice that this method has some similarities with the pattern presented at the beginning of the *Limiting concurrency* section. It essentially starts as many tasks from the queue as possible, without exceeding the concurrency limit. When each task is complete, it updates the count of running tasks and then starts another round of tasks by asynchronously invoking `next()` again. The interesting property of the `TaskQueue` class is that it allows us to dynamically add new items to the queue. The other advantage is that, now, we have a central entity responsible for the limitation of the concurrency of our tasks, which can be shared across all the instances of a function's execution. In our case, it's the `spider()` function, as you will see in a moment.

Refining the TaskQueue

The previous implementation of `TaskQueue` is sufficient to demonstrate the queue pattern, but in order to be used in real-life projects, it needs a couple of extra features. For instance, how can we tell when one of the tasks

has failed? How do we know whether all the work in the queue has been completed?Let's bring back some of the concepts we discussed in *Chapter 3*, *Callbacks and Events*, and let's turn the `TaskQueue` into an `EventEmitter` so that we can emit events to propagate task failures and to inform any observer when the queue is empty.The first change we have to make is to import the `EventEmitter` class and let our `TaskQueue` extend it:

```
import { EventEmitter } from 'node:events'
export class TaskQueue extends EventEmitter {
  constructor (concurrency) {
    super()
    // ...
  }
  // ...
}
```

At this point, we can use `this.emit` to fire events from within the `TaskQueue` `next()` method:

```
next () {
  if (this.running === 0 && this.queue.length ===
    return this.emit('empty')
  }
  while (this.running < this.concurrency && this.
    const task = this.queue.shift()
```

```
      task((err) => { // (2)
        if (err) {
          this.emit('error', err)
        }
        this.running--
        process.nextTick(this.next.bind(this))
      })
      this.running++
    }
  }
```

Comparing this implementation with the previous one, there are two additions here:

- Every time the `next()` function is called, we check that no task is running and whether the queue is empty. In such a case, it means that the queue has been drained and we can fire the `empty` event.
- The completion callback of every task can now be invoked by passing an error. We check whether an error is actually passed, indicating that the task has failed, and in that case, we propagate such an error with an `error` event.

Notice that in case of an error, we are deliberately keeping the queue running. We are not stopping other tasks in progress, nor removing any pending tasks. This is quite common with queue-based systems. Errors are expected to happen and rather than letting the system crash on these occasions, it is generally better to identify errors and to think about retry or re-

covery strategies. We will discuss these concepts a bit more in *Chapter 13*, *Messaging and Integration Patterns*.

> *Hypothetically, if you wanted to gracefully halt all tasks upon encountering an error, you might consider introducing a mechanism that signals the* `TaskQueue` *to stop accepting and processing new tasks. Imagine a scenario where we add a* `this.stopped` *flag to the* `TaskQueue` *class. In case of an error, this flag is immediately set to* `true`*, and the* `queue` *is cleared. This would prevent any further tasks waiting in the queue from being initiated. We could also add logic to stop accepting new tasks (e.g. by throwing an error in the* `pushTask()` *method when* `this.stopped` *is* `true`*) that could be pushed into the* `queue`*. While this approach would prevent the queue from starting any new tasks, it's important to note that it wouldn't be able to stop tasks that are already running or waiting on some asynchronous call to complete.*

Web spider version 4

Now that we have our generic queue to execute tasks in a limited concurrent flow, let's use it straightaway to refactor our web spider application.We are going to use an instance of `TaskQueue` as a work backlog; every URL that we want to crawl needs to be appended to the queue as a task. The starting URL will be added as the first task, then every other URL discovered during the crawling process will be added as well. The queue will man-

age all the scheduling for us, making sure that the number of tasks in progress (that is, the number of pages being downloaded or read from the filesystem) at any given time is never greater than the concurrency limit configured for the given `TaskQueue` instance.We have already defined the logic to crawl a given URL inside our `spider()` function. We can consider this to be our generic crawling task. For more clarity, it's best to rename this function `spiderTask`:

```
function spiderTask(url, maxDepth, queue, cb) { /
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      // error checking the file
      return cb(err)
    }
    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        // ignoring non-HTML resources
        return cb()
      }
      return readFile(filename, 'utf8', (err, fi
        if (err) {
          // error reading the file
          return cb(err)
        }
        spiderLinks(url, fileContent, maxDepth, c
        return cb()
```

```
      })
    }
    // The file does not exist, download it
    download(url, filename, (err, fileContent) =>
      if (err) {
        // error downloading the file
        return cb(err)
      }
      // if the file is an HTML file, spider it
      if (filename.endsWith('.html')) {
        spiderLinks(url, fileContent.toString('ut
        return cb()
      }
      // otherwise, stop here
      return cb()
    })
  })
}
```

Other than renaming the function, you might have noticed that we applied some other small changes:

1. The function signature now accepts a new parameter called `queue`. This is an instance of `TaskQueue` that we need to carry over to be able to append new tasks when necessary.
2. The function responsible for adding new links to crawl is `spiderLinks()`, so we need to make sure that we pass the queue instance when we call this function after downloading a new page.

In just a moment, we'll see that the `spiderLinks()` function will be simplified to a synchronous function, as it will only need to append tasks to the queue. We won't need to pass the callback to it anymore; instead, we'll just call `return cb()` after invoking it.Now, let's take a look at the `spiderLinks()` function. This function can be significantly simplified since it no longer has to track task completion—the queue now handles that. Its role will effectively become synchronous; it simply needs to call the new `spider()` function (which we'll define shortly) to push a new task to the queue for each discovered link:

```
function spiderLinks(currentUrl, body, maxDepth,
  if (maxDepth === 0) {
    return
  }
  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return
  }
  for (const link of links) {
    spider(link, maxDepth - 1, queue)
  }
}
```

Let's now revisit the `spider()` function, which needs to act as the *entry point* for the first URL; it will also be used to add every new discovered URL to the `queue`:

```
const spidering = new Set() // (1)
export function spider(url, maxDepth, queue) {
  if (spidering.has(url)) {
    return
  }
  spidering.add(url)
  queue.pushTask(done => { // (2)
    spiderTask(url, maxDepth, queue, done)
  })
}
```

As you can see, this function now has two main responsibilities:

1. It manages the bookkeeping of the URLs already visited or in progress by using the `spidering` set.
2. It pushes a new task to the `queue`. Once executed, this task will invoke the `spiderTask()` function, effectively starting the crawling of the given URL.

Finally, we can update the `spider-cli.js` script, which allows us to invoke our spider from the command line:

```
import { spider } from './spider.js'
import { TaskQueue } from './TaskQueue.js'
const url = process.argv[2] // (1)
const maxDepth = Number.parseInt(process.argv[3],
```

```
const concurrency = Number.parseInt(process.argv[
const spiderQueue = new TaskQueue(concurrency) //
spiderQueue.on('error', console.error)
spiderQueue.on('empty', () => console.log('Downlo
spider(url, maxDepth, spiderQueue) // (3)
```

This script is now composed of three main parts:

1. CLI arguments parsing. Note that the script now accepts a third additional parameter that can be used to customize the concurrency level.
2. A `TaskQueue` object is created, and listeners are attached to the `error` and `empty` events. When an error occurs, we simply want to print it. When the queue is empty, that means that we've finished crawling the website.
3. Finally, we start the crawling process by invoking the `spider` function.

After we have applied these changes, we can try to run the spider module again. When we run the following command:

```
node spider-cli.js https://loige.co 1 4
```

We should notice that no more than four downloads will be active at the same time.With this final example, we've concluded our exploration of callback-based patterns.

Summary

At the start of this chapter, we mentioned that Node.js can be challenging due to its asynchronous nature, especially for developers coming from other platforms. However, as you've seen, you can make asynchronous APIs work to your advantage. The tools you've learned about are flexible and provide solid solutions to many common problems, while also allowing for different programming styles to suit individual preferences.We've also continued refactoring and improving our web crawler example throughout the chapter. When working with asynchronous code, it can sometimes be tricky to find the right balance between simplicity and effectiveness, so take your time to digest the concepts covered and experiment with them.Our journey with asynchronous Node.js programming is just beginning. In the next few chapters, you'll dive into other widely used techniques like promises and async/await. Once you're familiar with all these approaches, you'll be able to pick the best one for your needs—or even combine multiple techniques within the same project.Before moving on, we highly recommend giving the exercises below a try to solidify your understanding. They'll help you practice the key concepts and prepare you for the chapters ahead.

Exercises

1. **File concatenation**: Write the implementation of `concatFiles()`, a callback-style function that takes two or more paths to text files in the filesystem and a destination file:

```
function concatFiles (srcFile1, srcFile2, srcFile
                      dest, cb) {
  // ...
}
```

This function must copy the contents of every source file into the destination file, respecting the order of the files, as provided by the arguments list. For instance, given two files, if the first file contains *foo* and the second file contains *bar*, the function should write *foobar* (and not *barfoo*) in the destination file. Note that the preceding example signature is not valid JavaScript syntax: you need to find a different way to handle an arbitrary number of arguments. For instance, you could use the **rest parameters** syntax (`nodejsdp.link/rest-parameters`).

1. **List files recursively**: Write `listNestedFiles()` , a callback-style function that takes, as the input, the path to a directory in the local filesystem and that asynchronously iterates over all the subdirectories to eventually return a list of all the files discovered. Here is what the signature of the function should look like:

```
function listNestedFiles (dir, cb) { /* ... */ }
```

Bonus points if you manage to avoid callback hell. Feel free to create additional helper functions if needed.

1. **Recursive find**: Write `recursiveFind()`, a callback-style function that takes a path to a directory in the local filesystem and a keyword, as per the following signature:

```
function recursiveFind(dir, keyword, cb) { /* ...
```

The function must find all the text files within the given directory that contain the given keyword in the file contents. The list of matching files should be returned using the callback when the search is completed. If no matching file is found, the callback must be invoked with an empty array. As an example, test case, if you have the files `foo.txt`, `bar.txt`, and `baz.txt` in `myDir` and the keyword `'batman'` is contained in the files `foo.txt` and `baz.txt`, you should be able to run the following code:

```
recursiveFind('myDir', 'batman', console.log)
// should print ['foo.txt', 'baz.txt']
```

Bonus points if you make the search recursive (it looks for text files in any subdirectory as well). Extra bonus points if you manage to perform the search within different files and subdirectories concurrently but be careful to keep the number of concurrent tasks under control!

1. **Broken links checker**: Write a `checkBrokenLinks()` function that takes a URL and a maximum depth level and reports any broken links (links returning a 404 status code) it encounters during the crawling process. You could start from the code we wrote for the web crawler and modify it so that, instead of downloading the content of the pages, it only checks for the HTTP status of each link. If the status code is 404, it should log the URL of the broken link and continue crawling other links. Tip: You could try using the HEAD method instead of GET when checking for links, as it only fetches the headers, which can speed up the process and reduce bandwidth usage.