Node.js High Performance

Node.js is a tool written in C that allows you to use JavaScript on the server-side. High performance on a platform such as Node.js is to take advantage of every aspect of your hardware, helping memory management act at its best. Spot the memory leaks and solve them fast with Node.js by monitoring and stopping them before they become an issue.

Starting with performance analysis concepts and their importance in helping Node.js developers eliminate performance bottlenecks, this book will take you through development patterns to avoid performance penalties. You will learn the importance of garbage collection and its behavior, and discover how to profile your processor, allowing better performance and scalability.

Who this book is written for

This book is for Node.js developers who want a more in-depth knowledge of the platform to improve the performance of their applications. Whether you have a base Node.js background or you are an expert who knows the garbage collector and want to leverage it to make applications more robust, the examples in this book will benefit you.

What you will learn from this book

- Develop applications using well-defined and well-tested development patterns
- Explore memory management and garbage collection to improve performance
- Monitor memory changes and analyze heap snapshots
- Profile CPU and improve your code to avoid patterns that force intensive processor usage
- Understand the importance of data and when you should cache information
- Educate yourself to always test your code and benchmark when needed
- Extend your application's scope and know what other elements can influence performance

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction and Composition'
- A synopsis of the book’s content
- More information on **Node.js High Performance**
Diogo Resende is a passionate developer obsessed with perfection in everything he works on. He loves everything about the Internet of Things, which is the ability to connect everything together and always be connected to the world.

He studied computer science and graduated in engineering. At that time, he deepened his knowledge of computer networking and security, software development, and cloud computing. Over the past 10 years, Diogo has embraced different challenges to develop applications and services to connect people with embedded devices around the world, building a bridge between old and uncommon protocols and the Internet of today.

ThinkDigital has been his employer and a major part of his life for the last few years. It offers services and expertise in areas such as computer networking and security, automation, smart metering, and fleet management and intelligence. Diogo has also published many open source projects. You can find them all, with an MIT license style, on his personal GitHub page under the username dresende.
High performance on a platform such as Node.js means knowing how to take advantage of every aspect of your hardware and helping memory management act at its best and correctly decide how to architect a complex application. Do not panic if your application starts consuming a lot of memory. Instead, spot the leak and solve it fast. Better yet, monitor and stop it before it becomes an issue.

What this book covers

Chapter 1, *Introduction and Composition*, introduces the subject, emphasizing performance analysis and the importance of benchmarking. It's about splitting applications into several smaller components, reducing the complexity of each component to a manageable level for the developers involved in the application. Here, you understand the importance of developing methodologies to break complexity into smaller and reusable modules that can more easily be analyzed and exchanged with other new and better modules during the course of the application's life cycle.

Chapter 2, *Development Patterns*, is about good programming patterns that help avoid performance penalties or help find them. You'll value the importance of carefully choosing techniques and patterns that are simple, and avoid future problems. With this in mind, you'll better understand how the language works, the importance of knowing the event loop, how asynchronous programming works best, and some of the first-class citizens of the language—streams and buffers.

Chapter 3, *Garbage Collection*, covers GC, its importance, and its behavior. Here, you get to understand V8 memory management, dead memory, and memory leaks. You also learn how to profile an application and spot memory leaks caused by bad programming where a developer hasn't deferenced objects correctly.
Chapter 4, *CPU Profiling*, is about profiling the processor and understanding why and when your application hogs your host. In this chapter, you understand the limits of the language and how to develop applications that can be divided into several components running across different hosts, allowing better performance and scalability.

Chapter 5, *Data and Cache*, explains externally stored application data and how it can affect your application's performance. It's about data stored locally in the application, the disk, a local service, a local network service or even the client host. In this chapter, you get to know that different types of data storage methods have different penalties, and these must be considered when choosing the best one. You learn that data can be stored locally or remotely and access to the data can be—and should be—cached sometimes, depending on the importance of the data.

Chapter 6, *Test, Benchmark, and Analyze*, is about testing and benchmarking applications. It's also about enforcing code coverage to avoid unknown application test zones. Then we cover benchmarks and benchmark analytics. You get to understand how good tests can pinpoint where to benchmark and analyze specific parts of the application to allow performance improvements.

Chapter 7, *Bottlenecks*, covers limits outside the application. This chapter is about the situations when you realize that the performance limit is not because of the application programing but external factors, such as the host hardware, network or client. You'll become aware of the limits that external components can impose on the application, locally or remotely. Moreover, the chapter explains that sometimes, the limits are on the client side and nothing can be done to improve the current performance.
Introduction and Composition

High performance is hard, and it depends on many factors. Best performance should be a constant goal for developers. To achieve it, a developer must know the programming language they use and, more importantly, how the language performs under heavy loads, these being disk, memory, network, and processor usage.

Developers will make the most out of a language if they know its weaknesses. In a perfect world, since every job is different, a developer should look for the best tool for the job. But this is not feasible and a developer wouldn't be able to know every best tool, so they have to look for the second best tool for every job. A developer will excel if they know few tools but master them.

As a metaphor, a hammer is used to drive nails, and you can also use it to break objects apart or forge metals, but you shouldn't use it to drive screws. The same applies to languages and platforms. Some platforms are very good for a lot of jobs but perform really badly at other jobs. This performance can sometimes be mitigated, but at other times, can't be avoided and you should look for better tools.

Node.js is not a language; it's actually a platform built on top of V8, Google's open source JavaScript engine. This engine implements ECMAScript, which itself is a simple and very flexible language. I say "simple" because it has no way of accessing the network, accessing the disk, or talking to other processes. It can't even stop execution since it has no kind of exit instruction. This language needs some kind of interface model on top of it to be useful. Node.js does this by exposing a (preferably) nonblocking I/O model using libuv. This nonblocking API allows you to access the filesystem, connect to network services and execute child processes.

The API also has two other important elements: buffers and streams. Since JavaScript strings are Unicode friendly, buffers were introduced to help deal with binary data. Streams are used as simple event interfaces to pass data around. Buffers and streams are used all over the API when reading file contents or receiving network packets.
A stream is a module, similar to the network module. When loaded, it provides access to some base classes that help create readable, writable, duplex, and transform streams. These can be used to perform all sorts of data manipulation in a simplified and unified format.

The buffers module easily becomes your best friend when converting binary data formats to some other format, for example, JSON. Multiple read and write methods help you convert integers and floats, signed or not, big endian or little endian, from 8 bits to 8 bytes long.

Most of the platform is designed to be simple, small, and stable. It’s designed and ready to create some high-performance applications.

**Performance analysis**

Performance is the amount of work completed in a defined period of time and with a set of defined resources. It can be analyzed using one or more metrics that depend on the performance goal. The goal can be low latency, low memory footprint, reduced processor usage, or even reduced power consumption.

The act of performance analysis is also called **profiling**. Profiling is very important for making optimized applications and is achieved by instrumenting either the source or the instance of the application. By instrumenting the source, developers can spot common performance weak spots. By instrumenting an application instance, they can test the application on different environments. This type of instrumentation can also be known by the name **benchmarking**.

Node.js is known for being fast. Actually, it’s not that fast; it’s just as fast as your resources allow it. What Node.js is best at is not blocking your application because of an I/O task. The perception of performance can be misleading in Node.js applications. In some other languages, when an application task gets blocked—for example, by a disk operation—all other tasks can be affected. In the case of Node.js, this doesn’t happen—usually.

Some people look at the platform as being single threaded, which isn't true. Your code runs on a thread, but there are a few more threads responsible for I/O operations. Since these operations are extremely slow compared to the processor’s performance, they run on a separate thread and signal the platform when they have information for your application. Applications blocking I/O operations perform poorly. Since Node.js doesn’t block I/O unless you want it to, other operations can be performed while waiting for I/O. This greatly improves performance.
Chapter 1

V8 is an open source Google project and is the JavaScript engine behind Node.js. It's responsible for compiling and executing JavaScript, as well as managing your application's memory needs. It is designed with performance in mind. V8 follows several design principles to improve language performance. The engine has a profiler and one of the best and fast garbage collectors that exist, which is one of the keys to its performance. It also does not compile the language into byte code; it compiles it directly into machine code on the first execution.

A good background in the development environment will greatly increase the chances of success in developing high-performance applications. It's very important to know how dereferencing works, or why your variables should avoid switching types. Here are other useful tips you would want to follow. You can use a style guide like JSCS and a linter like JSHint to enforce them to for yourself and your team. Here are some of them:

- Write small functions, as they're more easily optimized
- Use monomorphic parameters and variables
- Prefer arrays to manipulate data, as integer-indexed elements are faster
- Try to have small objects and avoid long prototype chains
- Avoid cloning objects because big objects will slow the operations

**Monitoring**

After an application is put into production mode, performance analysis becomes even more important, as users will be more demanding than you were. Users don't accept anything that takes more than a second, and monitoring the application's behavior over time and over some specific loads will be extremely important, as it will point to you where your platform is failing or will fail next.

Yes, your application may fail, and the best you can do is be prepared. Create a backup plan, have fallback hardware, and create service probes. Essentially, anticipate all the scenarios you can think of, and remember that your application will still fail. Here are some of those scenarios and aspects that you should monitor:

- When in production, application usage is of extreme importance to understand where your application is heading in terms of data size or memory usage. It's important that you carefully define source code probes to monitor metrics—not only performance metrics, such as requests per second or concurrent requests, but also error rate and exception percentage per request served. Your application emits errors and sometimes throws exceptions; it's normal and you shouldn't ignore them.
Introduction and Composition

• Don't forget the rest of the infrastructure. If your application must perform at high standards, your infrastructure should too. Your server power supply should be uninterruptible and stable, as instability will degrade your hardware faster than it should.

• Choose your disks wisely, as faster disks are more expensive and usually come in smaller storage sizes. Sometimes, however, this is actually not a bad decision when your application doesn't need that much storage and speed is considered more important. But don't just look at the gigabytes per dollar. Sometimes, it's more important to look at the gigabits per second per dollar.

• Also, your server temperature and server room should be monitored. High temperatures degrades performance and your hardware has an operation temperature limit. Security, both physical and virtual, is also very important. Everything counts for the standards of high performance, as an application that stops serving its users is not performing at all.

Getting high performance
Planning is essential in order to achieve the best results possible. High performance is built from the ground up and starts with how you plan and develop. It obviously depends on physical resources, as you can't perform well when you don't have sufficient memory to accomplish your task, but it also depends greatly on how you plan and develop an application. Mastering tools will give much better performance chances than just using them.

Setting the bar high from the beginning of development will force the planning to be more prudent. Some bad planning of the database layer can really downgrade performance. Also, cautious planning will cause developers to think more about use cases and program more consciously.

High performance is when you have to think about a new set of resources (processor, memory, storage) because all that you have is exhausted, not just because one resource is. A high-performance application shouldn't need a second server when a little processor is used and the disk is full. In such a case, you just need bigger disks.

Applications can't be designed as monolithic these days. An increasing user base enforces a distributed architecture, or at least one that can distribute load by having multiple instances. This is very important to accommodate in the beginning of the planning, as it will be harder to change an application that is already in production.
Most common applications will start performing worse over time, not because of deficit of processing power but because of increasing data size on databases and disks. You'll notice that the importance of memory increases and fallback disks become critical to avoiding downtime. It's very important that an application be able to scale horizontally, whether to shard data across servers or across regions.

A distributed architecture also increases performance. Geographically distributed servers can be more closed to clients and give a perception of performance. Also, databases distributed by more servers will handle more traffic as a whole and allow DevOps to accomplish zero downtime goals. This is also very useful for maintenance, as nodes can be brought down for support without affecting the application.

**Testing and benchmarking**

To know whether an application performs well or not under specific environments, we have to test it. This kind of test is called a benchmark. Benchmarking is important to do and it's specific to every application. Even for the same language and platform, different applications might perform differently, either because of the way in which some parts of an application were structured or the way in which a database was designed.

Analyzing the performance will indicate bottleneck of your application, or if you may, the parts of the application that perform not good as others. These are the parts that need to be improved. Constantly trying to improve the worst performing parts will elevate the application's overall performance.

There are plenty of tools out there, some more specific or focused on JavaScript applications, such as benchmarkjs (http://benchmarkjs.com/) and ben (https://github.com/substack/node-ben), and others more generic, such as ab (http://httpd.apache.org/docs/2.2/programs/ab.html) and httpload (https://github.com/perusio/httpload). There are several types of benchmark tests depending on the goal, they are as follows:

- **Load testing** is the simplest form of benchmarking. It is done to find out how the application performs under a specific load. You can test and find out how many connections an application accepts per second, or how many traffic bytes an application can handle. An application load can be checked by looking at the external performance, such as traffic, and also internal performance, such as the processor used or the memory consumed.
• Soak testing is used to see how an application performs during a more extended period of time. It is done when an application tends to degrade over time and analysis is needed to see how it reacts. This type of test is important in order to detect memory leaks, as some applications can perform well in some basic tests, but over time, the memory leaks and their performance can degrade.

• Spike testing is used when a load is increased very fast to see how the application reacts and performs. This test is very useful and important in applications that can have spike usages, and operators need to know how the application will react. Twitter is a good example of an application environment that can be affected by usage spikes (in world events such as sports or religious dates), and need to know how the infrastructure will handle them.

All of these tests can become harder as your application grows. Since your user base gets bigger, your application scales and you lose the ability to be able to load test with the resources you have. It's good to be prepared for this moment, especially to be prepared to monitor performance and keep track of soaks and spikes as your application users start to be the ones responsible for continuously test load.

Composition in applications
Because of this continuous demand of performant applications, composition becomes very important. Composition is a practice where you split the application into several smaller and simpler parts, making them easier to understand, develop, and maintain. It also makes them easier to test and improve.

Avoid creating big, monolithic code bases. They don't work well when you need to make a change, and they also don't work well if you need to test and analyze any part of the code to improve it and make it perform better.

The Node.js platform helps you—and in some ways, forces you to—compose your code. Node.js Package Manager (NPM) is a great module publishing service. You can download other people's modules and publish your own as well. There are tens of thousands of modules published, which means that you don't have to reinvent the wheel in most cases. This is good since you can avoid wasting time on creating a module and use a module that is already in production and used by many people, which normally means that bugs will be tracked faster and improvements will be delivered even faster.

The Node.js platform allows developers to easily separate code. You don't have to do this, as the platform doesn't force you to, but you should try and follow some good practices, such as the ones described in the following sections.
Using NPM

Don’t rewrite code unless you need to. Take your time to try some available modules, and choose the one that is right for you. This reduces the probability of writing faulty code and helps published modules that have a bigger user base. Bugs will be spotted earlier, and more people in different environments will test fixes. Moreover, you will be using a more resilient module.

One important and neglected task after starting to use some modules is to track changes and, whenever possible, keep using recent stable versions. If a dependency module has not been updated for a year, you can spot a problem later, but you will have a hard time figuring out what changed between two versions that are a year apart. Node.js modules tend to be improved over time and API changes are not rare. Always upgrade with caution and don't forget to test.

Separating your code

Again, you should always split your code into smaller parts. Node.js helps you do this in a very easy way. You should not have files bigger than 5 kB. If you have, you better think about splitting it. Also, as a good rule, each user-defined object should have its own separate file. Name your files accordingly:

```javascript
// MyObject.js
module.exports = MyObject;

function MyObject() {
    // …
}
MyObject.prototype.myMethod = function () { … };
```

Another good rule to check whether you have a file bigger than it should be; that is, it should be easy to read and understand in less than 5 minutes by someone new to the application. If not, it means that it’s too complex and it will be harder to track and fix bugs later on.

Remember that later on, when your application becomes huge, you will be like a new developer when opening a file to fix something. You can’t remember all of the code of the application, and you need to absorb a file behavior fast.
Embracing asynchronous tasks

The platform is designed to be asynchronous, so you shouldn't go against it. Sometimes, it can be really hard to make some recursive tasks or even simply cycle through a list of tasks that have to run serially. You should avoid creating a module to handle asynchronous tasks, as there are some used and tested by hundreds of thousands of people out there. For instance, async is a simple and very practical way of helping the developer perform better, and the learning curve is very smooth:

```javascript
async.each(users, function (user, next) {
    // do something on each user object
    return next();
}, function (err) {
    // done!
});
```

This module has a lot of methods similar to the ones you find in the array object, such as map, reduce, filter, and each, but for iterating asynchronously. This is extremely useful when your application gets more complex and some user actions require some serialized tasks. Error handling is also done correctly and the execution stop is done as expected. The module helps run serial or parallel tasks.

Also, serial tasks that would usually enforce a developer to nest calls and enter the callback hell can simply be avoided. This is especially useful when, for example, you need to perform a transaction on a database with several queries involved.

Another common mistake when writing asynchronous code is throwing errors. Callbacks are called outside the scope where they are defined, and so you cannot just put the callback inside a try/catch block. Therefore, avoid doing this unless it's a very critical error that should make your application stop and quit. In Node.js, throwing an exception without catching it will trigger an uncaughtException event.

The platform has a rule that is consensual for most developers—the so-called error-first callback style. This rule is of extreme importance, since it allows an easier reuse of your code. Even if you have a function where there's no chance of throwing an error, or when you just don't want it to throw and use some kind of error handling inside the function, your callback should always reserve the first argument for an error event if it's always null. This will allow your function to be used with an async module. Also, other developers will be counting on this style when debugging, so always reverse the first argument as an error object.

Plus, you should always reserve the last argument of the function as the callback. Never define arguments after your callback:

```javascript
function mySuperFunction(arg1, ..., argN, next) {
    // do some voodoo
```
Using library functions

Library functions are another type of module you should use. They help in handling repetitive tasks, and every developer has to perform such tasks. Some of these repetitive tasks can be done with no effort, just by using a library function from lodash or underscore. They are an important part of your code and have good optimizations that you don't even have to think about. Many cycling tasks, such as finding an object in an array based on an object key, or mapping an array of objects to an array of keys of every object, are one-liners in these libraries. Read the documentation first to avoid using the library and not fully using its potential.

Although these kinds of modules can be useful, they can also downgrade performance if they are not chosen well. Some modules are designed to help developers in some tasks, but do not target performance—just convenience. In other words, these modules can help you develop faster, but you shouldn't forget the complexity of each function. Otherwise, you will be calling the same function several times because you forget about its complexity, instead of calling it once and saving the results.

Using function rules

Functions are very important in this platform. This is no surprise since the language is functional and has first-class functions. There are some rules you should follow when writing functions that will make your life easier when debugging or optimizing it later. They also avoid some errors as they try to enforce some common structure. Once again, you can enforce these rules using, for example, JSCS (http://jscs.info/):

1. Always name your functions, especially when they're closures used as callbacks. This allows you to identify them in stack traces when your code breaks. Also, they allow a new developer to rapidly know what the function is supposed to do. Still, avoid long names:

```javascript
socket.on("data", function onSocketData(data) {
    // ...
});
```
2. Don't nest your conditions, and return as early as possible. If you have a condition that must return something in a function and if you return, you don't have to use the else statement. You also avoid a new indent level, reducing your code and simplifying its revision. If you don't do this, you will end up in a condition hell, with several levels if you have two or more conditions to satisfy:

```javascript
// do this
if (someCondition) {
    return false;
}
return someThing;
```

```javascript
// instead of this:
if (someCondition) {
    return false;
} else {
    return someThing;
}
```

3. Create small and simple functions. Don't span your functions for more lines than your screen can handle. Even if your task cannot be reused, split the function into smaller ones. It is even better to put it into a new module and publish it. In this way, you can reuse them at the frontend if you need them. This can also allow the engine to optimize some smaller functions when it is unable to optimize the previous big function. Again, this is important if you don't want a developer to be reading your application code for a week or two before being able to touch anything.

### Testing your modules

Testing your modules is a hard job and is usually neglected, but it's very important to make tests for your modules. The first ones are the hard ones. Look for a test tool that you like, such as vows, chai, or mocha. If you don't know how to start, read a module's documentation, or another module's test code. But don't give up on testing.

If you need help, read the test tools' websites mentioned earlier, as they usually help you get started. Alternatively, you can take a look at Igor's post [here](https://semaphoreci.com/community/tutorials/getting-started-with-node-js-and-mocha) at semaphore.
After you start adding one or two tests, more will follow. One big advantage of testing your module from the beginning is that when you spot a bug, you can make a test case for it, to be able to reproduce it and avoid it in the future.

Code coverage is not crucial but can help you see how your tests cover your module code base, and if you're just testing a small part. There are some coverage modules, such as *istanbul* or *jscoverage*; choose the one that works best for you. Code coverage is done together with testing, so if you don't test it, you won't be able to see the coverage.

As you might want to improve the performance of an application, every dependency module should be looked at for improvements. This can be done only if you test them. Dependency version management is of great importance, and it can be hard to keep track of new versions and changes, but they might give you some good news. Sometimes, modules are refactored and performance is boosted. A good example of this is database access modules.

**Summary**

Together, Node.js and NPM make a very good platform for developing high-performance applications. Since the language behind them is JavaScript and most applications these days are web applications, these combinations make it an even more appealing choice, as it's one less server-side language to learn (such as PHP or Ruby) and can ultimately allow a developer to share code on the client and server sides. Also, frontend and backend developers can share, read, and improve each other's code. Many developers pick this formula and bring with them many of their habits from the client side. Some of these habits are not applicable because on the server side, asynchronous tasks must rule as there are many clients connected (as opposed to one) and performance becomes crucial.

In the next chapter, we will cover some development patterns that help applications stay simple, fast, and scalable as more clients come along and start putting pressure on your infrastructure.
Where to buy this book

You can buy Node.js High Performance from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.