dynatrace

Feeling the love for Node.js

The performance basics you need to know now

Table of contents

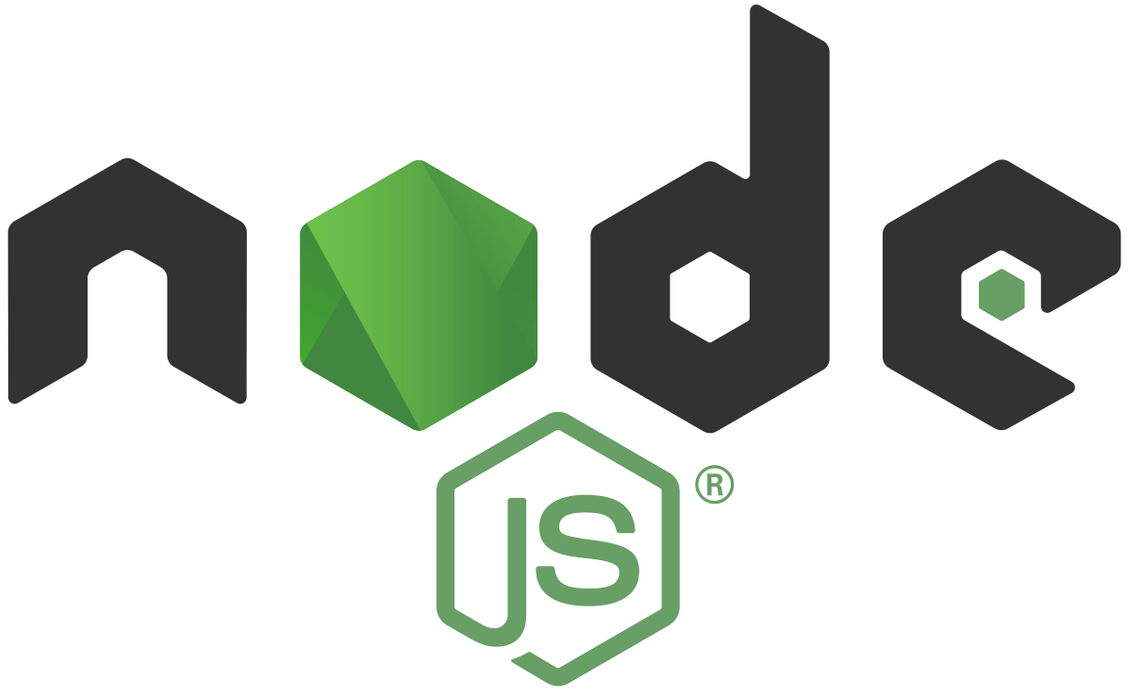
Executive summary

Node.js is not just new technology for the start-ups anymore. Enterprises of all types are beginning to use Node.js. But why?

- Functionality that is pragmatic and smart, offering basic building blocks instead of wizardry and black box functionality designed by framework developers.
- Program execution that doesn't stop and wait when performing I/O requests, which saves system resources and provides a better overall performance.
- Technology that bridges existing systems and new technologies and provides an easily deployable migration layer that either proxies data to existing systems or collects and reformats it for different uses.

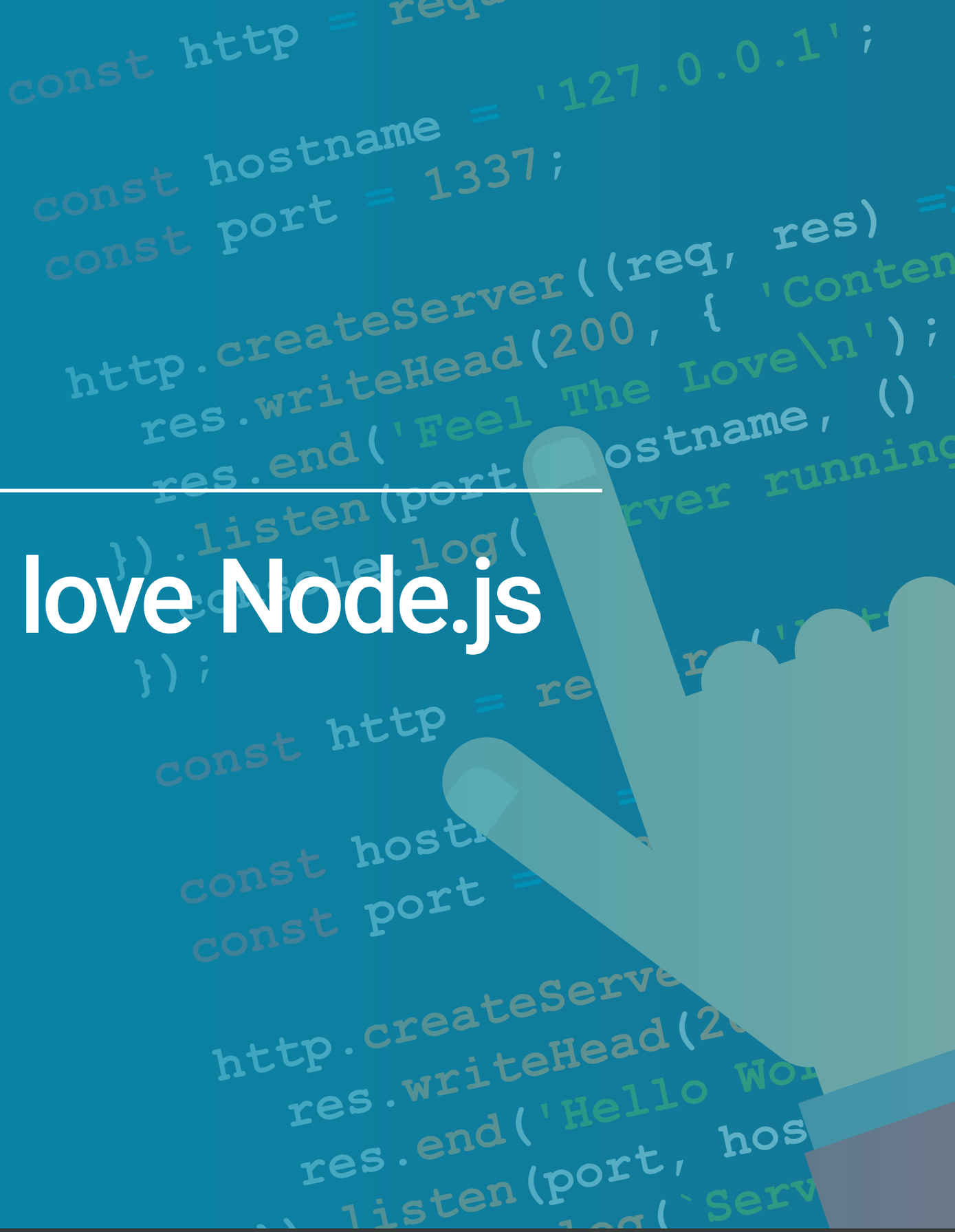
While Node.js has a lot of benefits, including a flat learning curve, the machinery that keeps it ticking is complex. You must understand it to avoid performance pitfalls.

In this eBook, we will discuss the basic building blocks of Node.js and how memory works for Node, including garbage collection. We will also give you some tips on how to hunt down memory leaks, CPU issues, and other problems to improve performance.



Chapter 1

Why enterprises love Node.js



Google searches and jobs

Seasoned developers know that every few months there is a new game-changing technology that passes into oblivion. This is not the case with Node.js.

Google Trends shows search traffic by keyword over time, and is a great way to identify the relevance of a given topic, especially when correlated with other terms. When you compare search traffic on Node.js to Ruby on Rails, you can clearly see that Node.js gained momentum after its introduction in 2009 and now has beaten its competitor.

Another indicator of relevance is *Indeed Job Trends.* If you compare job growth to other related technologies, you see Node.js again comes out ahead.

Alert readers may argue that the charts looked the same for technologies like Ruby on Rails some years back. So how's Node.js different?

Big established companies with years of legacy software are now shifting to Node.js.

What the big brands say

Every new kid on the technology block gains momentum in the beginning, especially because startups tend to use the freshest technologies.

But big established companies with years of legacy software are now shifting to Node.js – *Walmart, eBay, PayPal, Microsoft, The New York Times, Dow Jones, etc.* The list goes on.

It's uncommon for enterprises to adopt a technology like this so early in its hype cycle. We need to peek into how Node.js works to understand the reasons.

PayPal™

"Node.js powers our web applications and has allowed our teams to move much faster in bringing their designs to life."

-Jeff Harrell,
Director of Engineering at PayPal

"On the server side, our entire mobile software stack is completely built in Node. One reason was scale. The second is Node showed us huge performance gains."

-Kiran Prasad,
Mobile Development Lead at LinkedIn

LinkedIn™

ebay™

"Node's evented I/O model freed us from worrying about locking and concurrency issues that are common with multithreaded async I/O."

-Subbu Allarmarju,
Principal Member, Technical Staff at ebay

Black magic wizadry

For so many years, the trend has been toward framework and wizardry.

Just enter some commands in the shell you get a full-fledged ORM layer out of your database scheme. Apparently, it seemed like a good idea to abstract as much logic as possible away from the developer and offer some easy-to-use interface for functionality that framework developers *think* is useful.

You know those “build your own blog in five minutes” examples. Everything *looks* easy and great until you try to implement real requirements for real clients.

It’s tempting to take the easy road and rely on generators and out-of-the-box features, but this never works out. The time you save will be spent trying to figure out why the ORM layer logic hidden deep inside the framework doesn’t let you do a select on a date range. Instead of building from your own code, you end up spending your time reading through documents, Googling, and posting on StackOverflow just to understand code written by someone else.

Back to craftsmanship

Node.js is a framework based on JavaScript that uses Google's V8 engine.

It adds functionality in a very pragmatic and smart way by offering only the basic building blocks, like file system access, protocol support, and cryptology along with some basic utility functions.

Also of note, JavaScript supports event based programming — just look at the OnClick() in browser environments — and has functions as first class members. This makes it a perfect fit for event-callback constructs that are a cornerstone of asynchronous programming. In short, this means that program execution doesn't stop and wait when performing I/O requests, which saves system resources and provides better performance.

Enterprise grade LTS is another reason to love Node.js

A new Long Term Support model was introduced in the Fall, 2015.

Long-term support (LTS) releases will now be actively developed for 18 months and maintained for another 12 months. So a LTS release may effectively stay in production for 30 (!) months, while still receiving bug and security fixes.

For even more enterprise features and support, you can look up N|Solid.

"Reading Node.js docs feels like Zen: reduced, focused, and concise. It feels like getting back to the essence of programming again. No bells, no whistles."

– Daniel Khan,
Technology Strategist, Dynatrace

The Node.js bridge

During the last few years, the requirements for network-connected software systems changed dramatically.

> Responsive, dynamic sites replaced static or semi-static websites for desktop computers.

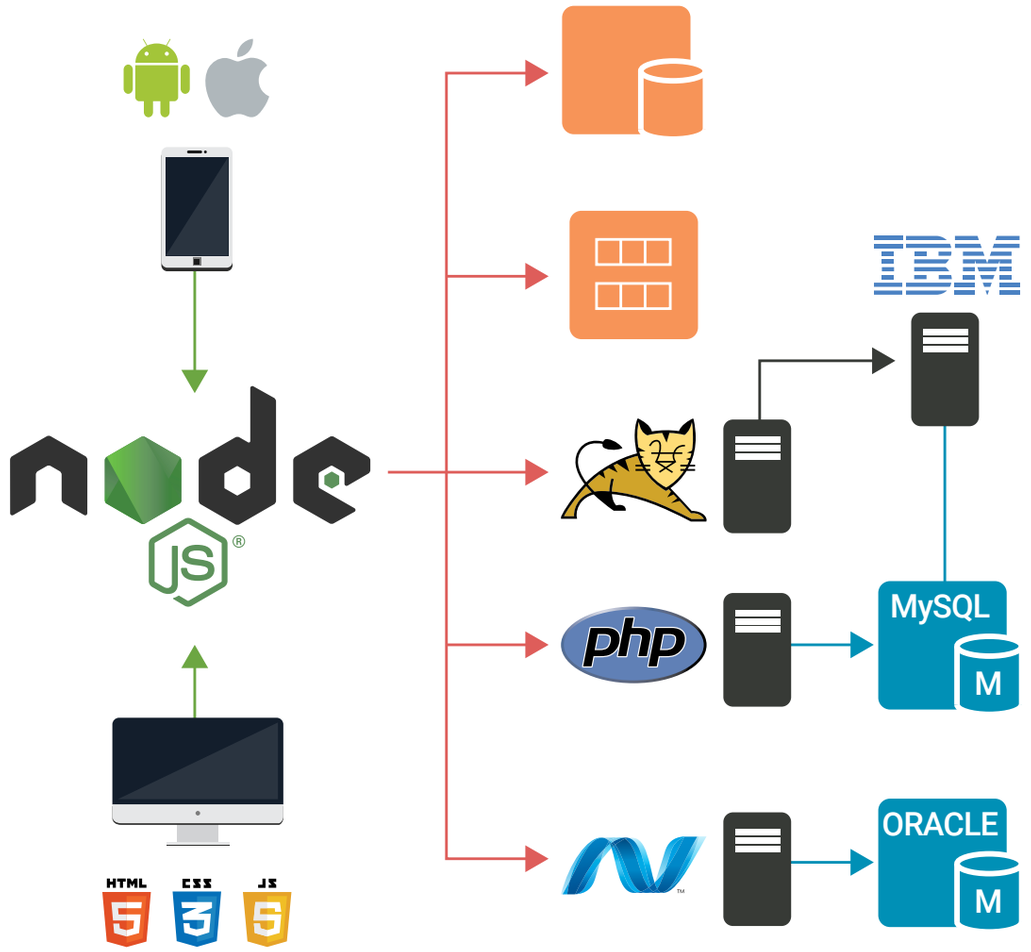
> Sites started to load additional content via Ajax/XHR or even rely on push technologies like WebSockets.

> Streaming services began to rise in popularity.

> Data of all kinds now needs to be delivered to an ever-growing number of users, with different devices in different formats.

These changing and growing requirements can either be fulfilled by adding new hardware and extending existing software or by adding a new layer that uses existing infrastructure more efficiently and supports the new technologies.

That's where Node.js shines — with modern, web-connected applications that need to gather data from different sources, consolidate it, and push it to many clients in real-time. It is the perfect bridging technology for existing systems because it provides an easily deployable migration layer that either proxies data to existing systems or collects and reformats data for different uses.



Chapter 2

The Node.js building blocks

Modules, packages, and immutable infrastructure

It can be a daunting task to implement a full-fledged web server when all you have is the basic HTTP commands.

But not relying on complex frameworks *does not mean that you have to build everything from scratch.*

Modules are an important building block for Node.js and expose functionality by the global data structure, *module.exports*.

Modules are included and assigned to a variable with *require()*. The exported functionality is only accessible through this variable, which means that every module creates a namespace.

The great thing about Node.js is that the Npm — the Node Package Manager, a collection of modules, currently holds more than 140,000 (!) packages for various purposes, and most of them are maintained on GitHub.

Many of them adhere to the *"Do one thing and do it well"* philosophy known from UNIX. The result is a pool of small bits of functionality that can be mixed and composed as needed. There are also a few larger packages that provide a set of functionality, with express being the most popular — it provides http, routing, and template rendering functions that most web projects need.

If you ever had to migrate a complex legacy php application to a new server, you know that this can get extremely unpleasant due to the globally installed extensions and libraries required. This is not the case with Node.js.

In Node.js, all dependencies are registered in *package.js* and installed into the projects directory structure. Spinning up a new instance running an application only takes minutes with Node.js because there are no dependencies to the environment (except the node binary) and no need for a specific server environment.

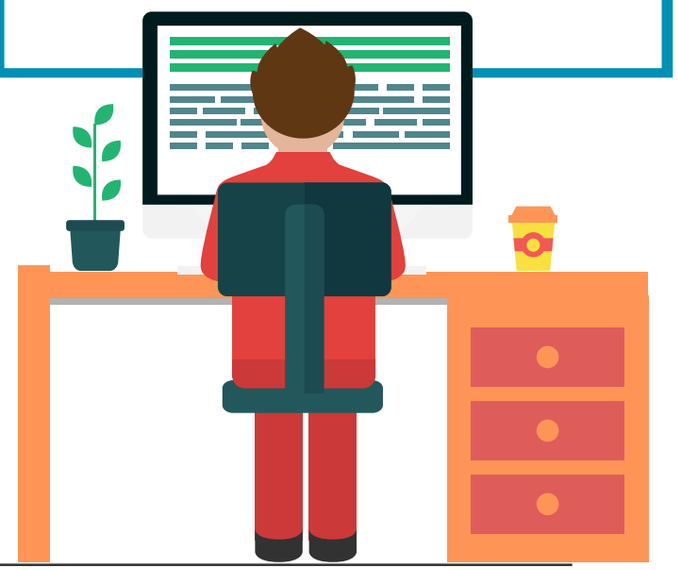
This makes Node.js the platform of choice for projects following the Immutable Infrastructure principle introduced by Chad Fowler in 2013.

Getting started

On Linux use your package manager to install the latest nodejs package. For all other operating systems, download the binaries from the Node.js website.

Then run Node.js applications by typing node <yourapplication.js>.

Here's a basic "Hello world" example. Run it and open http://localhost:4242 in your browser to see if everything is working.



Memory in Node.js means Google V8

While Node.js has a rather flat learning curve, the machinery that keeps it ticking is quite complex.

You must understand it to avoid performance pitfalls. Unlike platforms like PHP, Node.js applications are long-running processes. While this has a lot of positive implications such as allowing database connections to be set up once and then reused for all requests, this may also cause problems.

Node.js is a C++ program controlled via V8 JavaScript. Google V8 is a JavaScript engine initially created for Google Chrome, but it can also be used as a standalone. This makes it the perfect fit for Node.js. V8 compiles JavaScript down to native code and executes it, and manages memory allocation.

A running program is always represented through some space allocated in memory, the Resident Set. V8 uses a scheme similar to the Java Virtual Machine and divides the memory into segments.

- **Code:** the actual code being executed.
- **Stack:** contains all value types (primitives like integer or Boolean) with pointers referencing objects on the heap and pointers defining the control flow of the program.
- **Heap:** a memory segment dedicated to storing reference types like objects, strings, and closures.

Within Node.js, current memory usage can easily be queried by calling *process.memoryUsage()*.This function will return an object containing: Resident Set Size, Total Size of the Heap, Heap Used. You can use this function to record the memory usage over time in a heap graph.

The heap graph is highly volatile, but always stays within certain boundaries to keep the median consumption constant. The mechanism that allocates and frees heap memory is called *garbage collection*.

Understanding garbage collection

Every program that consumes memory requires a mechanism for reserving and freeing space.

If a program allocates memory that is never freed, the heap will constantly grow, creating a memory leak, until the usable memory is exhausted causing the program to crash.

Because Node.js JavaScript is compiled to native code by V8, the resulting native data structures don't have much to do with their original representation and are solely managed by V8. This means that we cannot actively allocate or de-allocate memory in JavaScript. V8 uses garbage collection to address this problem.

The theory behind garbage collection is simple: if a memory segment is not referenced from anywhere, we can assume that it is not used and therefore, it can be freed. However, retrieving and maintaining this information is complex because there may be chained references and indirections that form a complex graph structure.

Garbage collection is a costly process because it interrupts the execution of an application and impacts its performance. To remedy this situation, V8 uses two types of garbage collection:

> Scavenge — fast, but incomplete

> Mark-Sweep — relatively slow, but frees all non-referenced memory

Revisiting the data collected from *process.memoryUsage()* and the chart on the previous page, you can identify the different garbage collection types: the saw-tooth pattern is created by Scavenge runs and the downward jumps indicate Mark-Sweep operations.

By using the native module, node-gc-profiler, you can gather even more information about garbage collection runs. The module subscribes to garbage collection events fired by V8, and exposes them to JavaScript. The object returned indicates the type of garbage collection and the duration.

Chapter 3

Node.js and performance

Hunting down memory leaks

So if garbage collection cleans up the memory, why do you have to care at all? In fact, it is still possible — and easy — for memory leaks to suddenly appear in your logs.

Garbage collection tries its best to free memory, but on the chart (right), we see that consumption after a garbage collection run is constantly climbing, a clear indication of a leak.

Some causes of leaks are obvious — if you store data in process-global variables, like the IP of every visiting user in an array. Others are subtler like the famous Walmart memory leak that was caused by a tiny missing statement within Node.js core code, and which took weeks to track down.

V8 provides a way to dump the current heap, and the V8-profiler exposes this functionality to JavaScript. Here is the code to do this. There are more sophisticated approaches to detect anomalies, but this is a good place to start.

If there is a memory leak, you may end up with a significant number of heap dump files. So you should monitor this closely and add some alerting capabilities to that module. The same heap dump functionality is also provided within Chrome, and you can use Chrome developer tools to analyze the dump's V8-profiler.

One heap dump may not help you, however, because it won't show you how the heap develops over time. That's why Chrome developer tools allow you to compare different memory profiles.

By comparing two dumps, you get delta values that indicate which structures grew between two dumps.

The Node.js single thread and CPU issues

The statement 'Node.js runs in a single thread' is only partly true.

If you start up a simple application and look at the processes, you see that Node.js, in fact, spins up a number of threads. This is because Node.js maintains a thread pool to delegate synchronous tasks to, while Google V8 creates its own threads for tasks like garbage collection.

Node.js uses *libuv*, which provides the thread pool and an event loop to achieve asynchronous I/O.

Reading a file is a synchronous input output task and can take its time. In synchronous programming, the thread *runningreadfile()* would now pause and wait until the system call returns the contents of the file. That's why, for client-server environments (like web applications), most platforms will create one thread per request. A PHP application, within Apache using mod_php, works exactly like that.

The "single-threaded" Node.js works differently. To not block the main thread, it delegates the *readfile()* task to *libuv*. *Libuv* will then push the callback function to a queue and use some rules to determine how the task will be executed. In some cases, it will now use the thread pool to load off the work, but it may also create asynchronous tasks to be handled directly by the operating system.

The event loop continuously runs over the callback queue and will execute a callback, if the associated task has been finished. The execution of the callback — and this is very important — will again run on the main thread.

Usually this works just fine, but there will be a problem if someone decides to perform a CPU consuming operation like calculating some prime numbers inside the callback. This will block the main thread, prevent new requests from being processed and cause the application to slow down.

Mind your environment (aka *NODE_ENV*)

Most developers learn best by examples, which naturally tend to simplify matters and omit things that aren't essential for understanding.

This means that the "Hello World" example on Page 7, when used as starting point for an application, may be not suitable for production scenarios.

Environment variables are often used for distinguishing between production or development. Depending on those variables an application may turn debugging on or off, connect to a specific database, or listen on a specific port. In Node.js, *NODE_ENV* is used to set the current mode.

If you use Express.js and install the template engine without setting *NODE_ENV*, it will default to development. It will switch the view cache on, which makes sense. You don't want to restart your app to clear the cache every time you change some markup.

This means your views are parsed and rendered for every request in development mode. The question is whether this causes a notable performance hit.

It does. Setting *NODE_ENV* to production makes the application **three times faster.**

NODE_ENV works like any other environment variable (e.g. Setting *NODE_ENV PATH*) and setting it depends on your platform.

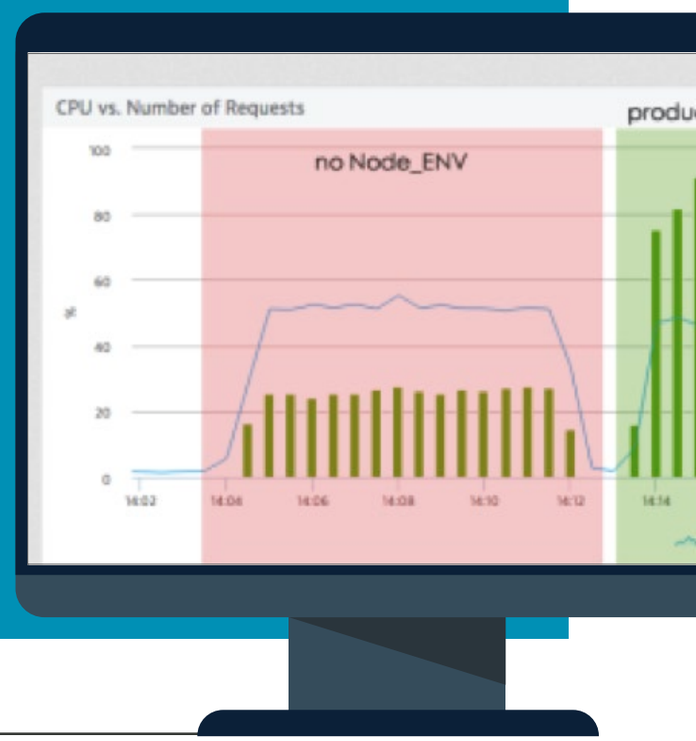
Linux and OSX:

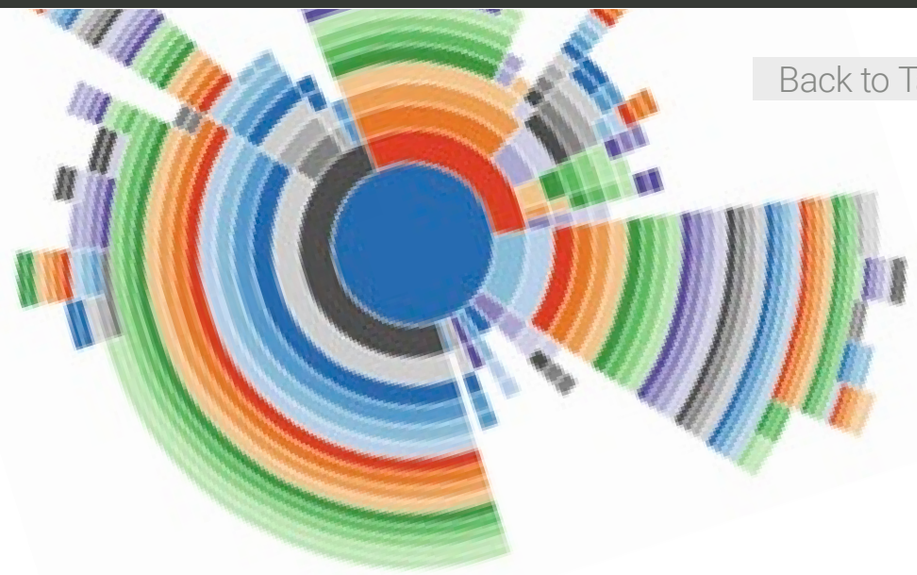
```
export NODE _ ENV=production
```

Windows:

```
SET NODE _ ENV=production
```

On all platforms, you can explicitly set it when starting your application like this: *NODE_ENV=production node myapp.js*. You may also use a module like *dotenv* to set it from an *.env* file in your application directory. You should avoid setting the environment directly in your code because this contradicts the purpose of environment variables.





Click here to open the chart in your browser and hover over the segments to display the function names.

CPU sampling and sunbursts

To trace down CPU heavy operations, you need a way to introspect on what's happening on the CPU in a given time period.

This introspection is called *CPU sampling* and luckily Google V8, the engine that compiles and executes JavaScript in Node.js, provides a CPU sampling interface. That can be accessed from JavaScript using the module v8-profiler.

This is the code you need to run the CPU profiler for five seconds every 30 seconds, and then store the sampling data under a timestamp-based filename. The file has a JSON tree structure, and there are a several ways to analyze it.

> Google Chrome ships with a set of developer tools useful for profiling and debugging JavaScript on websites. For a CPU intensive task, you can sort the call tree by heaviness — how much CPU time it used in the period — and quickly identify the issue. This approach works when there is an issue with a single function.

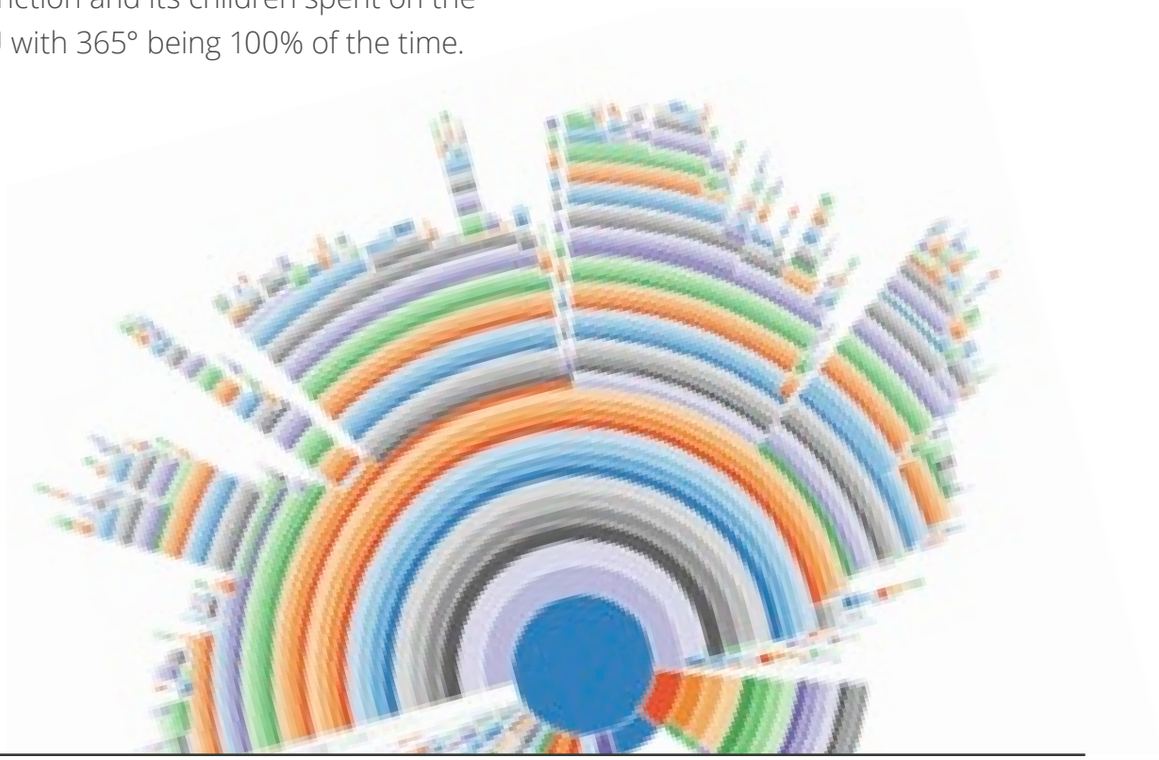
> When the problem is more complicated, Flame charts provide an eye-friendly visualization that presents the whole call stack. Here's a story by Netflix on how they used them to trace down a bug in their application.

> Another option, Sunburst charts do a particular good job of visualizing CPU problems.

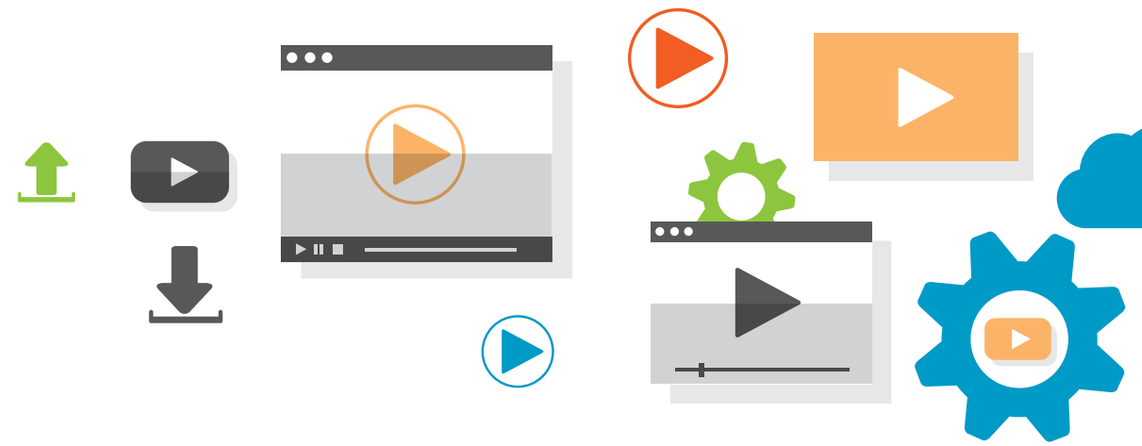
In Sunburst charts, the blue center is the main function. Every function called directly from main is placed in the adjacent row of the chart. If a function calls another function, it will be placed into an adjacent segment to the calling function.

One circle segment spans all its children and the distance from the center is the depth of a function within the call tree. The angle measure represents the time a function and its children spent on the CPU with 365° being 100% of the time.

To actually create one of these awesome charts, you can use D3.js, a popular and very powerful data visualization framework written in JavaScript. There are many code examples to show function name on a hover. You can see a full script here.



Additional resources



With Node.js, you, can gather debugging performance data via JavaScript using v8-profiler, and analyze the data in visualization tools like D3.js. For small, single tier applications this can provide a minimal viable monitoring solution.

To see the big picture for more complex applications, however, you need a complete performance monitoring solution. It should allow you to monitor the full Node.js stack — everything from V8 memory analysis to drill-down into database queries, code-level error diagnosis, user experience monitoring, outbound calls to external services and APIs, and server/infrastructure health analysis.

We hope this book has given you some insight on how enterprises are using Node.js and some tips on how to pinpoint performance problems. To the right are some additional resources on Node.js and performance monitoring in general.

Your homework

Videos

> Walmart Node.js memory leak

The blogroll

> About: Performance
> Joyent Blog on Walmart Node.js memory leak
> The New Stack
> The Netflix Tech Blog
> #monitoring life

Recommended reading

> Chetan Desai @ Intuit
> NodeSchool
> 7 performance metrics to release better software, faster
> DevOps hidden risks and how to achieve results