

# 11 Optimizing Performance

---

This chapter covers the following topics

- The performance optimization workflow
- Measuring a web applications performance
- Identifying hot code paths using flamegraphs
- Measuring the performance of a synchronous function
- Optimizing a synchronous function
- Measuring the performance of an asynchronous function
- Optimizing an asynchronous function

## Introduction

---

### Guest Author

This chapter was co-authored with Node.js Performance Guru and IOT expert Matteo Collina.

JavaScript runs in a single threaded event-loop. Node.js is a runtime built for evented I/O where multiple execution flows are processed concurrently but not in parallel. An example of this could be an HTTP server, tens of thousands of requests can be processed per second but only one instruction is being executed at any given time.

The performance of our application is tied to how fast we can process an individual execution flow prior to performing the next I/O operation.

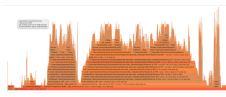
Through several recipes, this chapter demonstrates the Optimization Workflow as shown here:

## 1. Establish a **baseline**

```
node app.js  
autocannon -c 100 http://localhost:3000
```

## 2. Generate a **flamegraph**

```
0x app.js  
autocannon -c 100 http://localhost:3000
```



## 3. Identify the **bottleneck**

## 4. **Solve** the performance issue

## 5. **Verify** the solution

```
node app.js  
autocannon -c 100 http://localhost:3000
```

Repeat until  
satisfied

### *Optimization Workflow*

We'll be referencing the workflow throughout this chapter.

This chapter is about making our JavaScript code as fast as possible in order to increase I/O handling capacity, thus decreasing costs and increasing user experience.

## Benchmarking HTTP

Optimizing performance can be an endless activity. Our application can always be faster, more responsive and cheaper to run. However there's a trade off between developer time and compute time.

We can address the rabbit-hole nature of performance work in two steps. First we assess the current performance of an application, this is known as finding the baseline. Once the baseline is established we can set realistic goals based on our findings in the context of business requirements.

For instance, we find we can handle 200 requests per second but we wish to reduce server costs by one third. So we set a goal to reach 600 requests per second.

In this recipe, we'll be applying the first step in the optimization work-flow "Establish a baseline" to an HTTP server.

## Getting Ready

We will need the `autocannon` load testing tool.

So let's run the following command in our terminal:

```
$ npm install -g autocannon`
```

### About Autocannon

Autocannon is superior to other load testing tools in two main ways. Firstly it's cross-platform (macOS, Windows and Linux) whereas alternatives (such as `wrk` and `ab`) either do not run on Windows or are non-trivial to setup. Secondly, autocannon supports pipelining which allows for around 10% higher saturation than common alternatives

## How to do it

Let's create a small `express` application with a `/hello` endpoint.

First we'll create a folder with a `package.json` file and install Express:

```
$ mkdir http-bench
$ cd http-bench
$ npm init -y
$ npm install express --save
```

Now we'll create a `server.js` file with following content:

```
const express = require('express')
const app = express()

app.get('/hello', (req, res) => {
  res.send('hello world')
})

app.listen(3000)
```

We've created a server listening on port 3000, that exposes a `/hello` endpoint.

Now we'll launch it. On the command line we run:

```
$ node server
```

Next, if we open another terminal we can run a load test against our server and obtain a benchmark:

```
$ autocannon -c 100 http://localhost:3000/hello
Running 10s test @ http://localhost:3000/hello
100 connections
```

Stat	Avg	Stdev	Max
Latency (ms)	16.74	3.55	125
Req/Sec	5802.4	335.44	6083
Bytes/Sec	1.2 MB	73 kB	1.31 MB

```
58k requests in 10s, 12.19 MB read
```

Our results show an average of 5800 requests per second, with throughput of 1.2MB per second.

## The Optimization Workflow

When it comes to HTTP servers we should now know how to **establish a baseline**: by executing `autocannon` and generating a number in the form of req/sec (request per second).

The `-c 100` flag instructs `autocannon` to open 100 sockets and connect them to our server.

We can alter this number to whatever suits, but it's imperative that the connection count remains constant throughout an optimization cycle to avoid confounding comparisons between data sets.

## Load test duration

Duration defaults to 10 seconds but can be specified with the `-d` flag, followed by a number representing the amount of seconds to run the load test for. For instance `-d 20` will load the server for 20 seconds.

## How it works

The `autocannon` tool allocates a pool of connections (as per the `-c 100` setting), issuing a request on each socket immediately after the previous has completed.

This technique emulates a steady concurrency level whilst driving the target to maximum resource utilization without over saturating.

### Apache Benchmark



Apache Benchmark (`ab`) is another tool for load testing HTTP servers. However `ab` adopts a different paradigm, executes a specific amount of requests per second, regardless of whether prior requests have completed. Apache Benchmark can be used to saturate an HTTP endpoint to the point where some requests start to timeout, this can be useful for finding the saturation limit of a server but can also be problematic when it comes to troubleshooting a problem.

## There's more

Let's take a look at a common profiling pitfall, and learn how to benchmark POST requests.

### Profiling for Production

When measuring performance, we want the measurement to be relative to a production system. Modules may behave differently in development for convenience reasons, so being aware that this behavior can confound our results can prevent hours of wasted developer time.

Let's see how environment disparity can affect profiling output.

First we'll install Jade:

```
$ npm install -g jade
```



We cover Jade in detail in Chapter 6 Wielding Express

Next we'll update our `server.js` code

```
const express = require('express')
const path = require('path')
const app = express()

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.get('/hello', (req, res) => {
  res.render('hello', { title: 'Express' });
})

app.listen(3000)
```

We've setup Express to use the `views` folder for templates and use Jade to render them.

Let's create the views folder:

```
$ mkdir views
```

Finally we'll create a `views/hello.jade` file, with the following content:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    h1= title
```

Now we're ready to profile, first in one terminal we run the server:

```
node server.js
```

Now in another terminal window, we'll use `autocannon` to obtain a benchmark:

```
$ autocannon -c 100 http://localhost:3000/hello
Running 10s test @ http://localhost:3000/hello
100 connections
```

Stat	Avg	Stdev	Max
Latency (ms)	188.24	51.06	644
Req/Sec	526	80.76	583

```
Bytes/Sec    181.25 kB 28.34 kB 204.8 kB
```

```
5k requests in 10s, 1.82 MB read
```

That's a significant decrease in requests per second, only 10% of the prior rate in the main recipe. Are Jade templates really that expensive?

Not in production.

If we run our Express application in **production mode**, by setting the `NODE_ENV` environment variable to "production" we'll see results much closer to reasonable expectations.

Let's kill our server, then spin it up again like so:

```
NODE_ENV=production node server.js
```

Again, in a second terminal window we use `autocannon` to benchmark:

```
$ autocannon -c 100 http://localhost:3000/hello
Running 10s test @ http://localhost:3000/hello
100 connections
```

Stat	Avg	Stdev	Max
Latency (ms)	18.17	14.07	369
Req/Sec	5362.3	773.26	5867
Bytes/Sec	1.85 MB	260.17 kB	2.03 MB

```
54k requests in 10s, 18.55 MB read
```

Now results are much closer to those of our main recipe, around 90% of the performance of the `hello` route.

Running the application in production mode causes Express to make several production relevant optimizations.

In this case the increase in throughput is due to template caching.

In development mode (when `NODE_ENV` isn't explicitly set to production), Express will reload the template for every request which allows template changes without reloading the server.

Find out more about the production performance of Express at <http://expressjs.com/en/advanced/best-practice-performance.html#env>

## Measuring POST performance

The `autocannon` load tester can also profile POST request, we simply have to add a few flags.

Let's modify our `server.js` file so it can handle POST request at an endpoint we'll call `/echo`.

We change our `server.js` file to the following:

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: false}));

app.post('/echo', (req, res) => {
  res.send(req.body)
})

app.listen(3000)
```

We've removed our previous route, added in request body parser middleware and created an `/echo` route which mirrors the request body back to the client.

Now we can profile our `/echo` endpoint, using the `-m`, `-H` and `-b` flags:

```
$ autocannon -c 100 -m POST -H 'content-type=application/json' -b '{ "hello":
Running 10s test @ http://localhost:3000/echo
100 connections with 1 pipelining factor

Stat          Avg          Stdev        Max
Latency (ms)  25.77        4.8          156
Req/Sec       3796.1       268.95       3991
Bytes/Sec     850.48 kB   58.22 kB    917.5 kB

420k requests in 10s, 9.35 MB read
```

POST requests have roughly 65% the performance of GET requests when compared to the results from the main recipe.



## Loading the body from a file

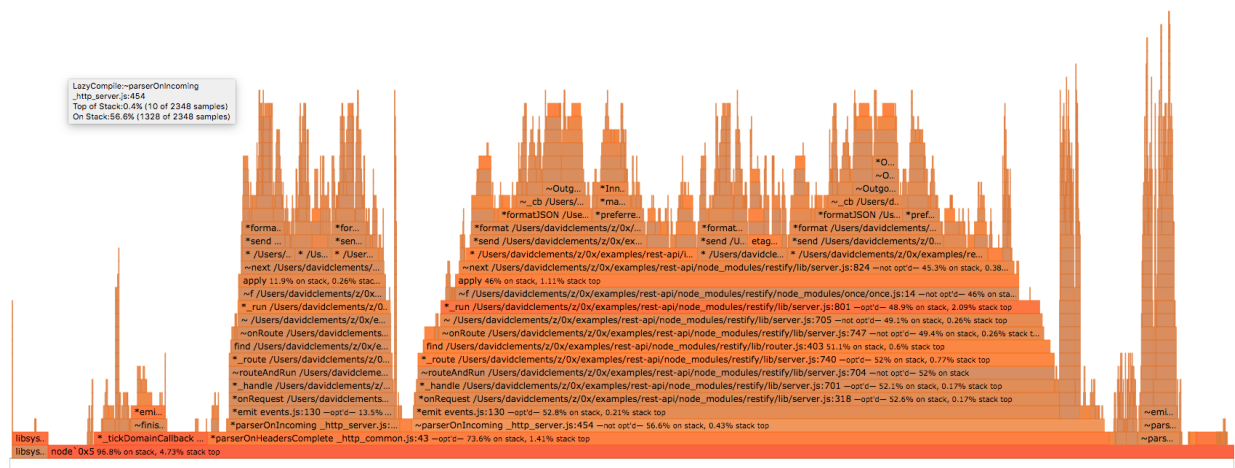


If we wish to get our POST body from a file, `autocannon` supports this via the `-i` flag.

## See also

- TBD

## Finding Bottlenecks with Flamegraphs



A flamegraph is an extremely powerful visual tool. It helps us to identify hot code paths in our application, and solve performance issues around those hot paths.

Flamegraphs compile stacks capturing during CPU profiling into a graphical representation that abstracts away the concept of time allowing us to analyze how our application works at a holistic level.


To put it another way, flamegraphs allow us to quickly determine how long each function (from C to JavaScript) has spent on the CPU, and which functions are causing the rest of the stack to be on CPU longer than it should be.

We're going to load-test a single route of an Express server, and use the `0x` flamegraphing tool to capture stacks and convert them into a flamegraph.

This recipe explores the second and third steps of the optimization workflow: "Generate a flamegraph" and "Identify the bottleneck".

## Getting Ready

In order to generate a flamegraph, we need Mac OS X (10.8 - 10.10)/macOS, a recent Linux distribution, or SmartOS.

Windows ! If we're using Windows, flamegraph tooling is limited, the best option is to install a virtual machine with Linux. See <http://www.storagecraft.com/blog/the-dead-simple-guide-to-installing-a-linux-virtual-machine-on-windows/> for details.

We'll also need to install `0x`, the flamegraph tool that can be installed as a global module:

```
$ npm install -g 0x
```

We'll also need to quickly scaffold an Express app, and efficient way to do this is install the `express-generator` module, and allow it generate an app for us.

## How to do it

Let's create a folder called `hello-server`, initialize a `package.json` and install `Express` and `Jade`:

```
$ mkdir hello-server
$ cd hello-server
$ npm init -y
$ npm install --save express jade
```

Now we'll create our `server.js` file

```
const express = require('express')
const path = require('path')
const app = express()

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.get('/hello', (req, res) => {
  res.render('hello', { title: 'Express' });
})

app.listen(3000)
```

Next, we'll create the views folder

```
$ mkdir views
```

Now we create a file in `views/hello.jade` , with the following content:

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    h1= title
```

Okay, now we're ready to profile the server and generate a flamegraph.

Instead of starting our server with the `node` binary, we use the globally installed `0x` executable.

We start our server with the following command:

```
0x server.js
```

Now we can use the `autocannon` benchmarking tool to generate some server activity.

## Autocannon

See previous recipe etc.

In another terminal window we use `autocannon` to generate load:

```
$ autocannon -c 100 http://localhost:3000/hello
Running 10s test @ http://localhost:3000/hello
100 connections with 1 pipelining factor

Stat      Avg      Stdev    Max
Latency (ms) 259.62  122.24  1267
Req/Sec      380.37  104.36  448
Bytes/Sec    131.4 kB 35.84 kB 155.65 kB

40k requests in 10s, 1.45 MB read
```

When the benchmark finishes, we hit CTRL-C in the server terminal. This will cause

`0x` to begin converting captured stacks into a flamegraph.

When the flamegraph has been generated a long URL will be printed to the terminal:

```
$ 0x server.js  
file:///path/to/profile-86501/flamegraph.html
```

The `0x` tool has created a folder named `profile-xxxx`, where `xxxx` is the PID of the server process.

If we open the `flamegraph.html` file with Google Chrome where we'll be presented with some controls, and a flamegraph resembling the following:



A flamegraph representing our `/hello` route under load

## 0x Theme

By default 0x presents flamegraphs with a black background, the flamegraph displayed here has a white background (for practical purposes). We can hit the "Theme" button (bottom left) to switch between black and white 0x themes.

## The Optimization Workflow

We should know now how to conduct step 2 of the Optimization Workflow laid in the introduction to this chapter: We launch the application with 0x to generate a flamegraph.

Functions that may be bottlenecks are displayed in darker shades of orange and red.

Hot spots at the bottom of the chart are usually less relevant to application and module developers, since they tend to relate to the inner workings of Node core. So if we ignore those, we can see that most of the hot areas appear within the two macro flames in the middle of the chart. A quick study of these show that many of the same functions appear within each - which means overall both stacks represent very similar logical paths.

## Graphical Reproducibility

We may find that our particular flamegraph doesn't exactly match the one included here. This is because of the non-deterministic nature of the profiling process. Overall however the general meaning of the flamegraph should be the same.

The right hand stack has a cluster of hot stacks some way up the main stack in the horizontal center of other diverging stacks.

TODO IMAGE HIGHLIGHTING PLACE MENTIONED

Let's click near the illustrated frame (or the equivalent identified stack if our current flamegraph is slightly different). Upon clicking the frame, 0x allows us to delve deeper by unfold the parent and child stacks to fill the screen, like so:



## Unfolded stacks

We should be able to see a pattern of darker orange stack frames, in each case the function is the same function appearing on line 48 of a `walk.js` file in one of our sub-dependencies. We have found our bottleneck.

## The Optimization Workflow



We've now covered step 3 of the Optimization Workflow: Find the bottleneck.

We've used the flamegraph structure and color coding to quickly understand where the slowest part of our server is.

## What's the cause?



Figured out what the root cause is? Check the There's more section of this recipe to find out!

## How it works

A flamegraph is generated by sampling the execution stack of our application during a benchmark.

A sample is a snapshot of all the functions being executed (nested) at the time it was taken.

It records the function that were currently executed by the CPU at that time, plus all the others that called it.

The sampled stacks are collected, and grouped together based on the functions called in them.

Each of those group is a "flame".

Flames have common function calls at various level of the stack.

Each line (Y axis) in a flame is a function call - known as a frame.

The width of each frame corresponds to the amount of time it was observed on the CPU. The time representation is an aggregate of all nested function calls.

For instance if function `A` calls function `B` the width of function `A` in the flamegraph will represent the time it took to execute both `A` and `B`.

If the frame is a darker shade of orange or red, then this particular function call was seen at the top of a stack more often than others.

If a stack frame is frequently observed at the top of the stack, it means that the function is preventing other instructions and I/O events from being processed.

In other words, it's blocking the event loop.

Each block representing a function call also contains other useful information, such as where the function is located in the code base, and if the function has been optimized or not by Node's JavaScript engine ([V8](#)).

## There's more

What's the underlying cause of our bottleneck, how does 0x actually profile our code, and how would we go about creating flamegraphs from production servers?

## Finding a solution

In this case Node.js is spending most of the execution time in the `acorn` dependency, which is a dependency of `jade`.

So we can conclude that template rendering is the bottleneck, our application is spending most of its time in parsing `.jade` files (through `acorn`).

In the previous recipe's **There's More** section we talked about **Profiling for Production**. Essentially if the server is in development mode templates are rendered each time whereas in production mode templates are cached. Our flamegraph has just made this abundantly clear.

Let's take a look at the resulting flamegraph generating from running the same

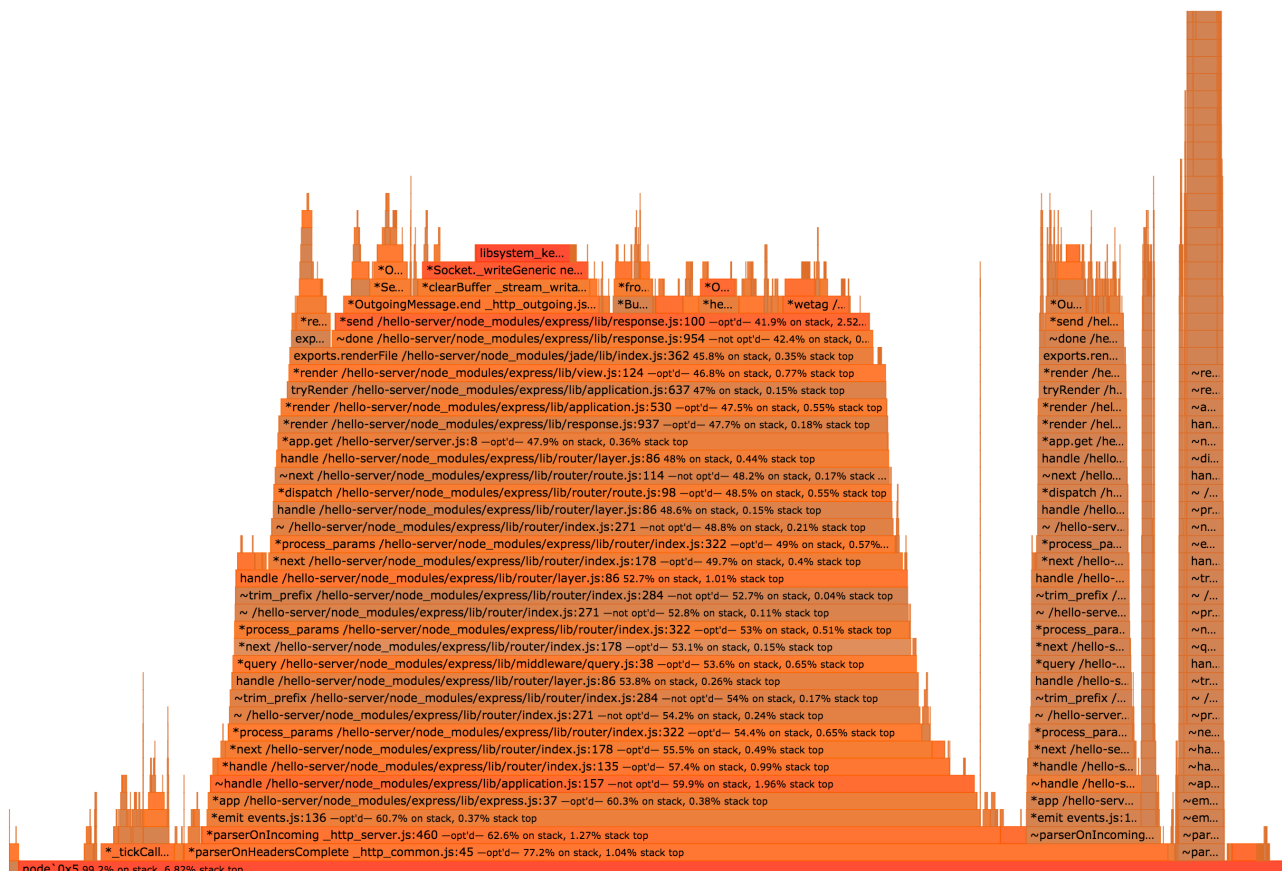


This time when we use `0x` to spin up our server we ensure `NODE_ENV` is set to production:

Now bench it with `autocannon` in another terminal window:

54k requests in 10s, 18.55 MB read

Now we hit CTRL-C on our server, once the flamegraph is generated it should look something like the following





This flamegraph has different shape it's much simpler. It has fewer functions and the hottest areas are where data is being written to a socket, this is the ideal since that's the exit point - that **should** be hot.

## How 0x works

When we use 0x to start our server, two processes are started. The first is node binary, 0x simply passes all non-0x arguments to node so our code is executed.

The second process is a system stack tracing tool ( perf in the case of Linux and dtrace in the case of macOS and SmartOS) to capture every function call in the C layer.

When 0x starts the node process, it adds a flag named --perf-basic-prof which is a V8 option (the JavaScript engine) which essentially enables the mapping of C++ V8 function calls to the corresponding JavaScript function calls.

The stacks are output to a folder, and in the case of macOS further mapping is applied. The stack output contains snapshots of the callstack for each 1 millisecond period the CPU was sampled. If a function is observed at the top of the stack for a particular snapshot (and it's not the ultimate parent of the stack) then it's taken a full millisecond on stack. If this happens multiple times, that's a strong indicator that it's a bottleneck.

To generate the flamegraph 0x processes these stacks into a JSON tree of parent and child stacks, then creates an HTML file containing the JSON plus a script that uses D3.js to visualize the parent child relationships and other meta-data in a flamegraph format.

## Flamegraphs and Production Servers

## See also

TBD

## Optimizing a synchronous function call

---

HTTP benchmarking and flamegraphs help us to understand our applications logical flow and rapidly pinpoint the areas which require optimization.

We're able to use the top-of-stack indicator (quickly recognized by darker orange or red areas in the flamegraph) along with some applied thought reasoning to pinpoint our performance issue to a specific synchronous function (or in some cases a group of synchronous functions).

Having covered Steps 1-3 of the Optimization Workflow (Establish a baseline, Generate a flamegraph, Identify the bottleneck) we will now venture into one permutation of Step 4, Solve the performance issue.

In this recipe we show how to isolate, profile and solve a synchronous function bottleneck.

## Getting Ready

Having understood the area of our code that needs work, our next step is to isolate the problem area and put together a micro-benchmark around it.

We'll be using [benchmark.js](#) to create our micro-benchmarks for single functions.

Let's create a new folder called `sync-opt`, initialize a `package.json` file and install the `benchmark` module as a development dependency:

```
$ mkdir sync-opt
$ npm init -y
$ npm install --save-dev benchmark
```

## How to do it

Let's assume that we've identified a bottleneck in our code base, and it happens to be a function called `divideByAndSum` which our flamegraph has revealed is appearing over 10% of time on the top of stack over multiple samples.

The function looks like this:

```
function divideByAndSum (num, array) {
  try {
    array.map(function (item) {
      return item / num
    }).reduce(function (acc, item) {
      return acc + item
    }, 0)
  } catch (err) {
```

```
    // to guard for division by zero
    return 0
  }
}
```

Our task now is to make this function faster.

The first step is to extract that function into its own module.

Let's create a file called `slow.js` :

```
function divideByAndSum (num, array) {
  try {
    array.map(function (item) {
      return item / num
    }).reduce(function (acc, item) {
      return acc + item
    }, 0)
  } catch (err) {
    // to guard for division by zero
    return 0
  }
}

module.exports = divideByAndSum
```

This is what an optimization candidate should look like. We've taken the function from the code base, put it in it's own file and exported it with `module.exports`.

The goal is to have an independent module, that we can benchmark in isolation.

We can now write a simple benchmark for it:

```
const benchmark = require('benchmark')
const slow = require('./slow')
const suite = new benchmark.Suite()

const numbers = []

for (let i = 0; i < 1000; i++) {
  numbers.push(Math.random() * i)
}

suite.add('slow', function () {
  slow(12, numbers)
})
```

```

suite.on('complete', print)

suite.run()

function print () {
  for (var i = 0; i < this.length; i++) {
    console.log(this[i].toString())
  }

  console.log('Fastest is', this.filter('fastest').map('name')[0])
}

```

Let's save this as `bench.js` and run our micro-benchmark to get a baseline:

```

$ node bench.js
slow x 11,014 ops/sec ±1.12% (87 runs sampled)
Fastest is slow

```

## Remove the use of the collections

The fastest way to iterate over an array is a for loop. Even though idiomatic Javascript prefers to use `forEach()`, `map()` and `reduce()`, iterating by calling a function is slower than a for loop.

We save the following module as `no-collections.js`:

```

function divideByAndSum (num, array) {
  var result = 0
  try {
    for (var i = 0; i < array.length; i++) {
      result += array[i] / num
    }
  } catch (err) {
    // to guard for division by zero
    return 0
  }
}

module.exports = divideByAndSum

```

We can then edit our `bench.js` file to add a test for this new module:

```

const noCollection = require('./no-collections')
suite.add('no-collections', function () {
  noCollection(12, numbers)
}

```

```
})
```

The results is impressive:

```
$ node bench.js  
slow x 11,014 ops/sec ±1.12% (87 runs sampled)  
no-collections x 58,190 ops/sec ±1.11% (87 runs sampled)  
Fastest is no-collections
```

## The danger of try-catch

Execptions are hard, and sometimes we must use a try/catch block. This is not the case:

```
function divideByAndSum (num, array) {  
  var result = 0  
  
  if (num === 0) {  
    return 0  
  }  
  
  for (var i = 0; i < array.length; i++) {  
    result += array[i] / num  
  }  
  
  return result  
}
```

Save this as `no-try-catch.js`, and then we edit our `bench.js` file to add a test for this new module:

```
const noTryCatch = require('./no-try-catch')  
suite.add('no-try-catch', function () {  
  noTryCatch(12, numbers)  
})
```

The results are even more relevant:

```
$ node bench.js  
slow x 11,014 ops/sec ±1.12% (87 runs sampled)  
no-collections x 58,190 ops/sec ±1.11% (87 runs sampled)  
no-try-catch x 231,196 ops/sec ±0.58% (92 runs sampled)  
Fastest is no-try-catch
```

---

We achieved a 20 times throughput improvement.

## How it works

Node.js is an evented I/O platform built on top of [V8](#), Google Chrome's Javascript VM. Node.js applications receive an I/O event (file read, data available on a socket, write completed), and then execute a Javascript callback. The next I/O event is processed after the Javascript function terminates. In order to write fast Node.js applications, our Javascript functions need to terminate as fast as possible.

[V8](#) offers two just-in-time compilers: one is activated when a function is loaded, and one when a function becomes "hot", and certain conditions are satisfied.

"Hot" functions are functions that either executed often or they take a long time to complete. If a function is not "hot", it will not show up in the flamegraph at the top of the stack, or in darker colors.

Both compiler generate machine code, but the optimizing compiler takes also into account the types and the code within the function. The rules that prevent a function to be optimized are called [V8 Optimization Killers](#).

## There's more

### Checking the optimization status

We can check if a function is optimized or optimizable by using the "V8 natives syntax", which we can turn on by executing our applications with `node --allow-natives-syntax app.js`.

We can then instrument the code like the following:

```
%GetOptimizationStatus(fn)
```

We can even write a little module to help us debugging these conditions:

```
function printStatus(fn) {  
  switch(%GetOptimizationStatus(fn)) {  
    case 1: console.log("Function is optimized"); break;  
    case 2: console.log("Function is not optimized"); break;  
    case 3: console.log("Function is always optimized"); break;  
  }
```

```

    case 4: console.log("Function is never optimized"); break;
    case 6: console.log("Function is maybe deoptimized"); break;
    case 7: console.log("Function is optimized by TurboFan"); break;
    default: console.log("Unknown optimization status"); break;
  }
}

module.exports = printStatus

```

We can then modify our `bench.js` file to verify that `no-try-collections.js` is optimized.

We can also see the optimization status of a function through the flamegraph generated by `0x`.

## Tracing optimization and deoptimization events

We can tap into the V8 decision process regarding when to optimize a function by running our code with `node --trace-opt --trace-deopt app.js`. We will get some lines that resemble this:

```

[marking 0x21e29c142521 <JS Function varOf (SharedFunctionInfo 0x1031e5bfa4b9)
[compiling method 0x21e29c142521 <JS Function varOf (SharedFunctionInfo 0x1031
[optimizing 0x21e29c142521 <JS Function varOf (SharedFunctionInfo 0x1031e5bfa4
[completed optimizing 0x21e29c142521 <JS Function varOf (SharedFunctionInfo 0x

```

In the above snippet, a function named `varOf` is marked for optimization and then it is optimized.

While reading the output, we would also notice lines like this:

```

[marking 0x21e29d33b401 <JS Function (SharedFunctionInfo 0x363485de4f69)> for
[compiling method 0x21e29d33b401 <JS Function (SharedFunctionInfo 0x363485de4f
[optimizing 0x21e29d33b401 <JS Function (SharedFunctionInfo 0x363485de4f69)> -

```

This is an anonymous function: it is really hard to know where this is defined. `0x` use several techniques to reconstruct the line of code where that is defined, but we can only access that if that line appears in the flamegraph. We must remember to always name our functions.

From time to time, we can also see a deoptimization happening:

```
[deoptimizing (DEOPT eager): begin 0x1f5a5b1fd601 <JS Function forOwn (SharedF
376]
    ;;; deoptimize at 109463: not a Smi
    reading input frame forOwn => node=3, args=13, height=2; inputs:
      0: 0x1f5a5b1fd601 ; (frame function) 0x1f5a5b1fd601 <JS Function
forOwn (SharedFunctionInfo 0x1f5a5b161259)>
      1: 0x1f5a5b1fa7d9 ; [fp + 32] 0x1f5a5b1fa7d9 <JS Function lodash
(SharedFunctionInfo 0x1f5a5b153b19)>
      2: 0x2e17991e6409 ; [fp + 24] 0x2e17991e6409 <an Object with map
0x366bf3f38b89>
      3: 0x2e17991ebb41 ; [fp + 16] 0x2e17991ebb41 <JS Function
(SharedFunctionInfo 0x1f5a5b1ca9e1)>
      4: 0x1f5a5b1eae1 ; [fp - 24] 0x1f5a5b1eae1 <FixedArray[272]>
      5: 0x1f5a5b1ee239 ; [fp - 32] 0x1f5a5b1ee239 <JS Function
baseForOwn (SharedFunctionInfo 0x1f5a5b155d39)>
    ...
```

The reason for the deoptimization is "not a SMI", which means that the function was expecting a 32-bit fixed integer and it got something else instead.

## See also

TBD

## Optimizing an asynchronous function

Node.js is an asynchronous runtime built for I/O heavy applications, and most of our code will involve some form of asynchronous callbacks. In the previous recipes we covered how to verify if there is a performance issue, where is the issue, and how to optimize a single Javascript function.

Some times, a performance bottleneck is part of an asynchronous flow, and it is hard to pinpoint where the performance issue is. In this recipe, we will cover that case in depth.

## Getting Ready

In this recipe, we will optimize an HTTP API built on [Express][expressjs] and MongoDB. We will use MongoDB version 3.2, which we will need to install from the MongoDB <https://www.mongodb.com> website or the package manager of our operating system.



Before starting, we will need to start MongoDB and then load some data through a little Node.js script. We save the following as `load.js` :

```
const MongoClient = require('mongodb').MongoClient
const url = 'mongodb://localhost:27017/test';
var count = 0
var max = 1000

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')

  function insert (err) {
    if (err) throw err

    if (count++ === max) {
      return db.close()
    }

    collection.insert({
      value: Math.random() * 1000000
    }, insert)
  }

  insert()
})
```

The above script depends on the `mongodb` module, which we should install via `npm i mongodb`. By running `node load.js` we will load 1000 entries into our MongoDB database.

## How to do it

We can write a very simple HTTP server that calculates the average of all the data points we have inserted.

```
const MongoClient = require('mongodb').MongoClient
const express = require('express')
const app = express()

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
```

```

app.get('/hello', (req, res) => {
  collection.find({}).toArray(function (err, data) {
    if (err) {
      res.send(err)
      return
    }
    const sum = data.reduce((acc, d) => acc + d.value, 0)
    const result = sum / data.length
    res.send('' + result)
  })
})

app.listen(3000)
})

```

We can save this server as `server.js`, and generate a benchmark:

```

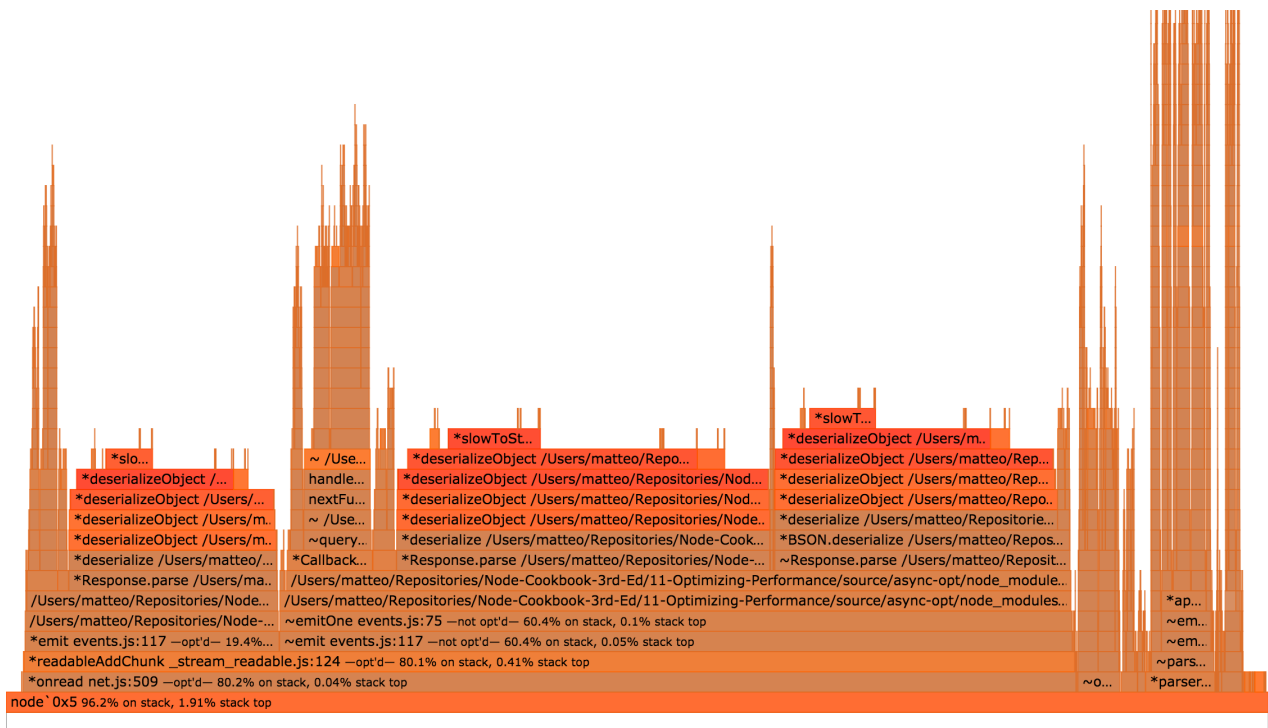
$ autocannon -c 1000 -d 5 http://localhost:3000/hello
Running 5s test @ http://localhost:3000/hello
1000 connections

Stat          Avg      Stdev    Max
Latency (ms) 2373.5   573.86   3352
Req/Sec       315.8    154.76   433
Bytes/Sec     68.02 kB 33.03 kB 94.21 kB

2k requests in 5s, 342.64 kB read
2 errors

```

We can now generate a flamegraph with `0x server.js` and `autocannon -c 1000 -d 5 http://localhost:3000/hello`.



In the above flamegraph, we can see that the darker areas are related to `deserializeObject` and `slowToString`. These are related to the amount of data being received from MongoDB. The *best* way to fix this issue would be to not doing this computation at all, and store (and update) the computed value whenever it changes. In some situations, this is not possible, and in this recipe we will focus on those.



In the flamegraph, there is a little "tower" of stacked function calls in the middle, if we zoom in by clicking on a function in there, we can see that there is some time spent into reduce. As we know, ES5 collections might causes slowdown, so we can rewrite our server as:

```

const MongoClient = require('mongodb').MongoClient
const express = require('express')
const app = express()

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
  app.get('/hello', (req, res) => {
    collection.find({}).toArray(function sum (err, data) {
      if (err) {
        res.send(err)
        return
      }
      var sum = 0
      const l = data.length
      for (var i = 0; i < l; i++) {
        sum += data[i].value
      }
      const result = sum / data.length
      res.send('' + result)
    })
  })

  app.listen(3000)
})

```

We can then run autocannon to see how it performs:

```

$ autocannon -c 1000 -d 5 http://localhost:3000/hello
Running 5s test @ http://localhost:3000/hello
1000 connections

```

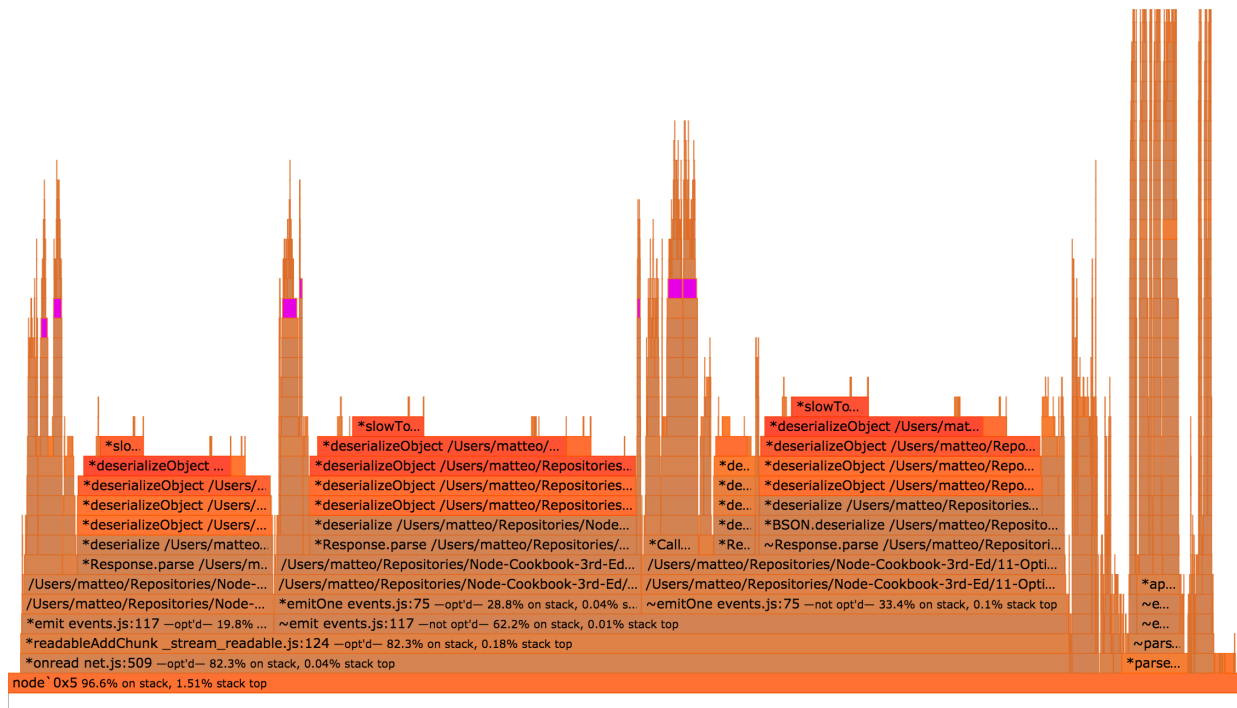
Stat	Avg	Stdev	Max
Latency (ms)	2293.1	569.38	3244
Req/Sec	331.8	142.23	456
Bytes/Sec	71.53 kB	30.94 kB	102.4 kB

```

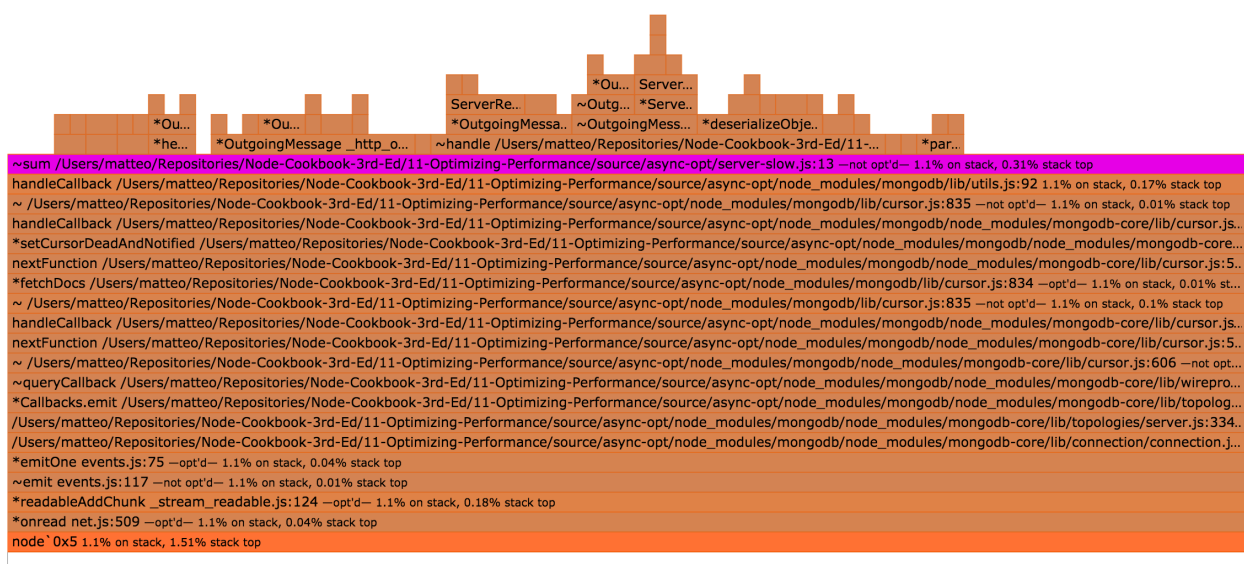
2k requests in 5s, 360 kB read
5 errors

```

We had a very small increase in throughput (5%). We can also generate a new flamegraph to see how our `sum` function is now performing.



Once we have generated a flamegraph with `0x`, we can use the `search` box on the top-right corner to locate `sum` function calls. If we click on one of functions, we get:



In the above detail of the flamegraph, we can see that the `sum` function was not optimized (not opt'd).

The `sum` function was not optimized because it is instantiated for every request, and then it need to be optimized by V8. However, it is only executed once, and it has no possibility of being optimized.

We can work around this problem by changing our `server.js` to:

```
const MongoClient = require('mongodb').MongoClient
```

```

const express = require('express')
const app = express()

var url = 'mongodb://localhost:27017/test';

function sum (data) {
  var sum = 0
  const l = data.length
  for (var i = 0; i < l; i++) {
    sum += data[i].value
  }
  return sum
}

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
  app.get('/hello', (req, res) => {
    collection.find({}).toArray(function (err, data) {
      if (err) {
        res.send(err)
        return
      }
      const result = sum(data) / data.length
      res.send('' + result)
    })
  })

  app.listen(3000)
})

```

We have extracted the actual iteration of the array into a top-level function that can be optimized by V8 and reused throughout the life of our process. Let's see how it performs:

```

$ autocannon -c 1000 -d 5 http://localhost:3000/hello
Running 5s test @ http://localhost:3000/hello
1000 connections

```

Stat	Avg	Stdev	Max
Latency (ms)	2164.37	526.15	2960
Req/Sec	359.4	138.31	457
Bytes/Sec	77.93 kB	29.74 kB	102.4 kB

2k requests in 5s, 389.95 kB read

From our starting point of 315 request per second, we have achieved a 14%

performance improvement just by optimizing a very hot for loop.

## How it works

Whenever we allocate a new function, it needs to be optimized by V8. The soonest V8 can optimize a new function, is after its first invocation. Node.js is built around callbacks and functions: when we need to wait for some I/O, we allocate a new function, wrapping the state in a closure. By using top-level functions for CPU-intensive behavior, we can assure we deliver amazing performance to our user.

## There's more

### A database solution

The recipe focuses on Javascript code that we can change, as some times we cannot change how the data is stored in our database easily. However, some times it is possible.

We can write another Node.js script to calculate our average, to be run whenever one of the data point changes:

```
const MongoClient = require('mongodb').MongoClient
const url = 'mongodb://localhost:27017/test';
var count = 0
var max = 1000

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
  const average = db.collection('averages')

  collection.find({}).toArray(function (err, data) {
    if (err) { throw err }
    average.insert({
      value: data.reduce((acc, v) => acc + v, 0) / data.length
    }, function (err) {
      if (err) { throw err }
      db.close()
    })
  })
})
})
```

Then, we can rewrite our server as:

```

const MongoClient = require('mongodb').MongoClient
const express = require('express')
const app = express()

var url = 'mongodb://localhost:27017/test';

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
  app.get('/hello', (req, res) => {
    collection.findOne({}, function sum (err, data) {
      res.send('' + data.value)
    })
  })

  app.listen(3000)
})

```

And finally, we can verify the throughput of this work:

```

$ autocannon -c 1000 -d 5 http://localhost:3000/hello
Running 5s test @ http://localhost:3000/hello
1000 connections

```

Stat	Avg	Stdev	Max
Latency (ms)	391.47	66.91	746
Req/Sec	2473.2	385.05	2849
Bytes/Sec	537.4 kB	82.99 kB	622.59 kB

12k requests in 5s, 2.68 MB read

Avoiding computation at all is the first solution for any performance issue.

## A caching solution

For high-performance applications, we might want to leverage in-process caching to save time for repeated CPU-bound tasks. We will use two modules for this:

[lru-cache][lru-cache] and `fastq`.

`lru-cache` implements an performant *least recently used* cache, where values are stored with a time to live. `fastq` is a performant queue implementation, to sequentialize the calls to compute the average. We want to fetch the data and compute the result once. Here is `server.js` implementing this behavior:



```

const MongoClient = require('mongodb').MongoClient
const express = require('express')
const LRU = require('lru-cache')
const fastq = require('fastq')
const app = express()

var url = 'mongodb://localhost:27017/test';

function sum (data) {
  var sum = 0
  const l = data.length
  for (var i = 0; i < l; i++) {
    sum += data[i].value
  }
  return sum
}

MongoClient.connect(url, function(err, db) {
  if (err) { throw err }
  const collection = db.collection('data')
  const queue = fastq(work)
  const cache = LRU({
    maxAge: 1000 * 5 // 5 seconds
  })

  function work (req, done) {
    const elem = cache.get('average')
    if (elem) {
      done(null, elem)
      return
    }
    collection.find({}).toArray(function (err, data) {
      if (err) {
        done(err)
        return
      }
      const result = sum(data) / data.length
      cache.set('average', result)
      done(null, result)
    })
  }

  app.get('/hello', (req, res) => {
    queue.push(req, function (err, result) {
      if (err) {
        res.send(err.message)
        return
      }
      res.send('' + result)
    })
  })
})

```

```
    })  
  
    app.listen(3000)  
  })
```

This is the best-performing solution so far:

```
$ autocannon -c 1000 -d 5 http://localhost:3000/hello  
Running 5s test @ http://localhost:3000/hello  
1000 connections
```

Stat	Avg	Stdev	Max
Latency (ms)	107.58	423.06	5024
Req/Sec	3660.4	1488.23	4675
Bytes/Sec	792.17 kB	321.93 kB	1.02 MB

```
18k requests in 5s, 3.97 MB read
```

## See also

TBD

## Recipe Title

---

### Getting Ready

### How to do it

### How it works

### There's more

## See also