NODE HERO

Get started with Node.js and deliver software products using it.

From the Engineers of

trace
by RisingStack

Table of contents

GETTING STARTED WITH NODE.JS

We are going to start with the basics - no prior Node.js knowledge is needed. The goal of this book is to get you started with Node.js and make sure you understand how to write an application using it.

In this very first chapter, you will learn what Node.js is, how to install it on your computer and how to get started with it - so in the next ones we can do actual development. Let's get started!

Node.js in a Nutshell

The official Node.js logo

> *Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an* ***event-driven, non-blocking I/O*** *model that makes it lightweight and efficient.*

In other words: Node.js offers you the possibility to write servers using JavaScript with an incredible performance. As the official statement says: Node.js is a runtime that uses the same V8 Javascript engine you can find in the Google Chrome browser. But that wouldn't be enough for Node.js's success - Node.js utilizes libuv, a multi-platform support library with a focus on asynchronous I/O.

The official libuv logo

From a developer's point of view Node.js is single-threaded - but under the hood **libuv handles threading, file system events, implements the event loop, features thread pooling** and so on. In most cases you won't interact with it directly.

Installing Node.js to get started

To get the latest Node.js binary you can visit the official Node.js website: https://nodejs.org/en/download/.

With this approach it is quite easy to get started - however if later down the road you want to add more Node.js versions, it is better to start using nvm, the Node Version Manager.

Once you install it, you can use a very simple CLI API that you can interact with:

Installing Node.js Versions

```
nvm install 4.4
```

Then, if you want to check out the experimental version:

```
nvm install 5
```

To verify that you have Node.js up and running, run this:

```
node --version
```

If everything is ok, it will return the version number of the currently active Node.js binary.

Using Node.js Versions

If you are working on a project supporting Node.js v4, you can start using it with the following command:

```
nvm use 4
```

Then you can switch to Node.js v5 with the very same command:

```
nvm use 5
```

Okay, now we know how to install and switch between Node.js versions - but what's the point?

Since the Node.js Foundation was formed, Node.js has a release plan. It's quite similar to the other projects of the Linux Foundation. This means that there are two releases: the stable release and the experimental one. In Node.js the stable versions with long-term support (LTS) are the ones starting with even numbers (4, 6, 8 ...) and the experimental version are the odd numbers (5, 7 ...).
We recommend you to use the LTS version in production and try out new things with the experimental one.

If you are on Windows, there is an alternative for nvm: nvm-windows.

Hello World

To get started with Node.js, let's try it in the terminal! Start Node.js by simply typing node:

```
$ node
>
```

Okay, let's try printing something:

```
$ node
> console.log('hello from Node.js')
```

Once you hit Enter, you will get something like this:

```
 > console.log('hello from Node.js')
hello from Node.js
undefined
```

Feel free to play with Node.js by using this interface - I usually try out small snippets here if I don't want to put them into a file.

It is time to create our Hello Node.js application!

Let's start with creating a file called `index.js`. Open up your IDE (Atom, Sublime, Code - you name it), create a new file and save it with the name `index.js`. If you're done with that, copy the following snippet into this file:

```
// index.js

console.log('hello from Node.js')
```

To run this file, you should open up your terminal again and navigate to the directory in which you placed index.js.

Once you successfully navigated yourself to the right spot, run your file using thenode `index.js` command. You can see that it will produce the same output as before - printing the string directly into the terminal.

Modularization of Your Application

Now you have your `index.js` file, so it is time to level up your game! Let's create something more complex by splitting our source code into multiple JavaScript files with the purpose of readability and maintainability. To get started, head back to your IDE (Atom, Sublime, Code - you name it) and create the following directory structure (with empty files), but leave the `package.json` out for now, we will generate it automatically in the next step:

```
├── app
|   ├── calc.js
|   └── index.js
├── index.js
└── package.json
```

Every Node.js project starts with creating a `package.json` file - you can think of it as a JSON representation of the application and its' dependencies. It contains your application's name, the author (you), and all the dependencies that is needed to run the application. *We are going to cover the dependencies section later in the* ***Using NPM*** *chapter of Node Hero.*

You can interactively generate your `package.json` file using the `npm init` command in the terminal. After hitting enter you will asked to give several inputs, like the name of your application, version, description and so on. No need to worry, just hit enter until you get the JSON snippet and the question `is it ok?`. Hit enter one last time and viola, your package.json has been automatically generated and placed in the folder of your application. If you open that file in your IDE, it will look very similar to the code snippet below.

```
// package.json
{
  "name": "@risingstack/node-hero",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
  "author": "",
  "license": "ISC"
}
```

It's a good practice to add a start script to your `package.json` - once you do that as shown in the example above you can start your application with the `npm start` command as well. It comes really handy when you want to deploy your application to a PaaS provider - they can recognize it and start your application using that.

Now let's head back to the first file you created called `index.js`. I recommend to keep the this file very thin - only requiring the application itself (the index.js file from the `/app` subdirectory you created before). Copy the following script into your `index.js` file and hit save to do this:

```
// index.js

require('./app/index')
```

Now it is time to start building the actual application. Open the index.js file from the /app folder to create a very simple example: adding an array of numbers. In this case the `index.js` file will only contain the numbers we want to add, and the logic that does the calculation needs to be placed in a separate module.

Paste this script to the `index.js` file in your `/app` directory.

```
// app/index.js
const calc = require('./calc')

const numbersToAdd = [
  3,
  4,
  10,
  2
]

const result = calc.sum(numbersToAdd)
console.log(`The result is: ${result}`)
```

Now paste the actual business logic into the `calc.js` file that can be found in the same folder.

```
// app/calc.js
function sum (arr) {
  return arr.reduce(function(a, b) {
    return a + b
  }, 0)
}

module.exports.sum = sum
```

To check if you'd succeeded, save these files, open up terminal and enter `npm start` or node `index.js`. If you did everything right, you will get back the answer: 19. If something went wrong, review the console log carefully and find the issue based on that.

In our next chapter called Using NPM we are going to take a look on how to use NPM, the package manager for JavaScript.

USING NPM

In this chapter, you'll learn what NPM is and how to use it. Let's get started!

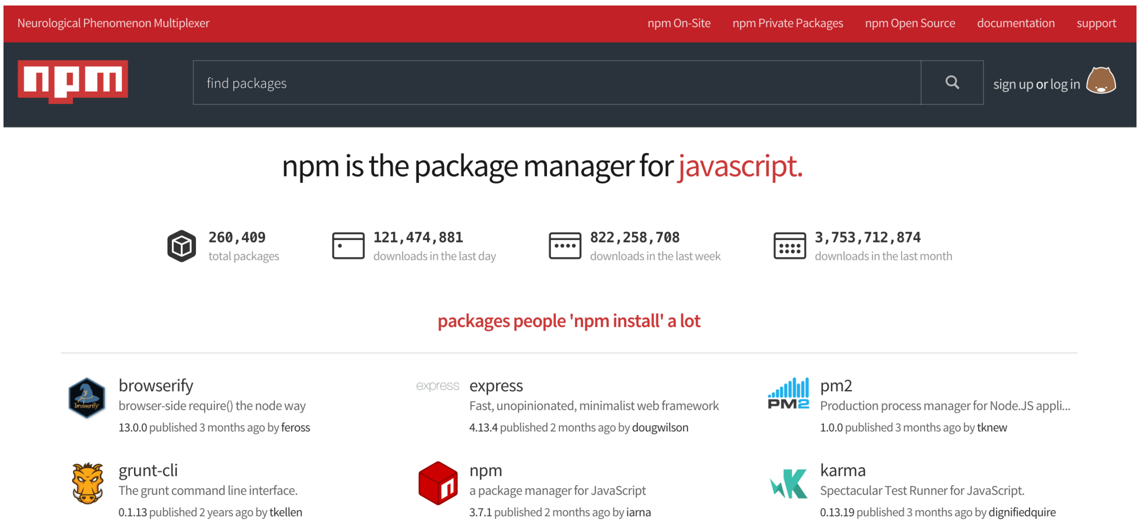
NPM in a Nutshell

NPM is the package manager used by Node.js applications - you can find a ton of modules here, so you don't have to reinvent the wheel. It is like Maven for Java or Composer for PHP. There are two primary interfaces you will interact with - the NPM website and the NPM command line toolkit.

Both the website and the CLI uses the same registry to show modules and search for them.

The Website

The NPM website can be found at https://npmjs.com. Here you can sign up as a new user or search for packages.



npm is the package manager for javascript.

260,409 total packages
121,474,881 downloads in the last day
822,258,708 downloads in the last week
3,753,712,874 downloads in the last month

packages people 'npm install' a lot

browserify
browser-side require() the node way
13.0.0 published 3 months ago by feross

express
Fast, unopinionated, minimalist web framework
4.13.4 published 2 months ago by dougwilson

pm2
Production process manager for Node.JS appli...
1.0.0 published 3 months ago by tknew

grunt-cli
The grunt command line interface.
0.1.13 published 2 years ago by tkellen

npm
a package manager for JavaScript
3.7.1 published 2 months ago by iarna

karma
Spectacular Test Runner for JavaScript.
0.13.19 published 3 months ago by dignifiedquire

The Command Line Interface

To run the CLI you can run it simply with:

```
npm
```

Note, that NPM is bundled with the Node.js binary, so you don't have to install it - however, if you want to use a specific npm version, you can update it. If you want to install npm version 3, you can do that with: `npm install npm@3 -g`.

Using NPM: Tutorial

You have already met NPM in the previous chapter, when you created the `package.json` file. Let's extend that knowledge!

Adding Dependencies

In this section you are going to learn how to add runtime dependencies to your application.

Once you have your package.json file you can add dependencies to your application. Let's add one! Try the following:

```
npm install lodash --save
```

With this single command we achieved two things: first of all, `lodash` is downloaded and placed into the `node_modules` folder. This is the folder where all your external dependencies will be put. Usually, you don't want to add this folder to your source control, so if you are using git make sure to add it to the `.gitignore file`.

This can be a good starting point for your `.gitignore` [Click here for the GitHub link]

Let's take a look at what happened in the `package.json` file! A new property called `dependencies` have appeared:

```
"dependencies": {
  "lodash": "4.6.1"
}
```

This means that lodash with version `4.6.1` is now installed and ready to be used. Note, that NPM follows SemVer to version packages.

> *Given a version number MAJOR.MINOR.PATCH, increment the **MAJOR** version when you make incompatible API changes, **MINOR** version when you add functionality in a backwards-compatible manner, and **PATCH** version when you make backwards-compatible bug fixes. For more information: http://semver.org/*

As `lodash` is ready to be used, let's see how we can do so! You can do it in the same way you did it with your own module except now you don't have to define the path, only the name of the module:

```
// index.js
const _ = require('lodash')

_.assign({ 'a': 1 }, { 'b': 2 }, { 'c': 3 });
// → { 'a': 1, 'b': 2, 'c': 3 }
```

Adding Development Dependencies

In this section you are going to learn how to add build-time dependencies to your application.

When you are building web applications, you may need to minify your JavaScript files, concatenating CSS files and so on. The modules that will do it will be only ran during the building of the assets, so the running application doesn't need them.

You can install such scripts with:

```
npm install mocha --save-dev
```

Once you do that, a new section will appear in your `package.json` file called `devDependencies`. All the modules you install with `--save-dev` will be placed there - also, they will be put in the very same `node_modules` directory.

NPM Scripts

NPM script is a very powerful concept - with the help of them you can build small utilities or even compose complex build systems.

The most common ones are the start and the `test` scripts. With the `start` you can define how one should start your application, while `test` is for running tests. In your `package.json` they can look something like this:

```
"scripts": {
    "start": "node index.js",
    "test": "mocha test",
    "your-custom-script": "echo npm"
  }
```

Things to notice here:

- `start`: pretty straightforward, it just describes the starting point of your application, it can be invoked with `npm start`
- `test`: the purpose of this one is to run your tests - one gotcha here is that in this case `mocha` doesn't need to be installed globally, as npm will look for it in the `node_modules/.bin folder`, and mocha will be placed there as well. It can be invoke§d with: `npm test`.
- `your-custom-script`: anything that you want, you can pick any name. It can be invoked with `npm run your-custom-script` - don't forget the `run` part!

Scoped / Private Packages

Originally NPM had a global shared namespace for module names - with more than 250.000 modules in the registry most of the simple names are already taken. Also, the global namespace contains public modules only.

NPM addressed this problem with the introduction of scoped packages. Scoped packages has the following naming pattern:

```
@myorg/mypackage
```

You can install scoped packages the same way as you did before:

```
npm install @myorg/mypackage --save-dev
```

It will show up in your `package.json` in the following way:

```
"dependencies": {
  "@myorg/mypackage": "^1.0.0"
}
```

Requiring scoped packages works as expected:

```
npm install @myorg/mypackage --save-dev
```

For more information refer to the NPM scoped module docs.

In the next chapter, you can **learn the principles of async programming** using callbacks and Promises.

UNDERSTANDING ASYNC PROGRAMMING

In this chapter, I'll guide you through async programming principles, and show you how to do async in JavaScript and Node.js.

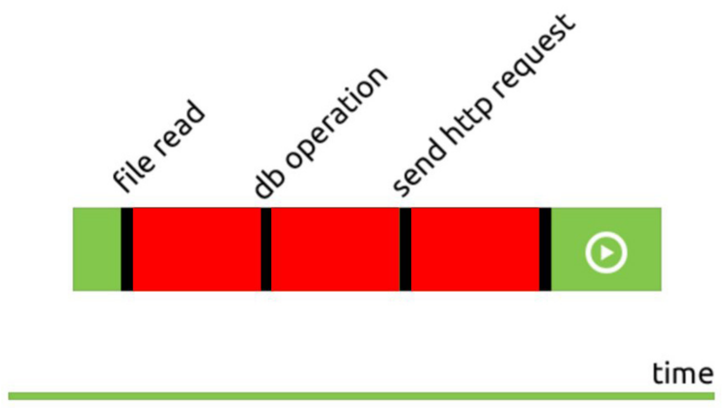
Synchronous Programming

In traditional programming practice, most I/O operations happen synchronously. If you think about Java, and about how you would read a file using Java, you would end up with something like this:

```
try(FileInputStream inputStream = new
FileInputStream("foo.txt")) {
    Session IOUtils;
    String fileContent = IOUtils.toString(inputStream);
}
```

What happens in the background? The main thread will be blocked until the file is read, which means that nothing else can be done in the meantime. To solve this problem and utilize your CPU better, you would have to manage threads manually.

If you have more blocking operations, the event queue gets even worse:



(The ***red bars*** *show when the process is waiting for an external resource's response and is blocked, the* ***black bars*** *show when your code is running, the* ***green bars*** *show the rest of the application)*

To resolve this issue, Node.js introduced an asynchronous programming model.

Asynchronous programming in Node.js

> *Asynchronous I/O is a form of input/output processing that permits other processing to continue before the transmission has finished.*

In the following example, I will show you a simple file reading process in Node.js - both in a synchronous and asynchronous way, with the intention of show you what can be achieved by avoiding blocking your applications.

Let's start with a simple example - reading a file using Node.js in a synchronous way:

```
const fs = require('fs')
let content
try {
  content = fs.readFileSync('file.md', 'utf-8')
} catch (ex) {
  console.log(ex)
}
console.log(content)
```

What did just happen here? We tried to read a file using the synchronous interface of the `fs` module. It works as expected - the `content` variable will contain the content of `file.md`. The problem with this approach is that Node.js will be blocked until the operation is finished - meaning it can do absolutely nothing while the file is being read.

Let's see how we can fix it!

Asynchronous programming - as we know now in JavaScript - can only be achieved with functions being first-class citizens of the language: they can be passed around like any other variables to other functions. **Functions that can take other functions as arguments are called higher-order functions.**

One of the easiest example for higher order functions:

```
const numbers = [2,4,1,5,4]

function isBiggerThanTwo (num) {
  return num > 2
}

numbers.filter(isBiggerThanTwo)
```

In the example above we pass in a function to the filter function. This way we can define the filtering logic.

This is how callbacks were born: if you pass a function to another function as a parameter, you can call it within the function when you are finished with your job. No need to return values, only calling another function with the values.

These so-called error-first callbacks are in the heart of Node.js itself - the core modules are using it as well as most of the modules found on NPM.

```
const fs = require('fs')
fs.readFile('file.md', 'utf-8', function (err, content) {
  if (err) {
    return console.log(err)
  }

  console.log(content)
})
```

Things to notice here:

- **error-handling:** instead of a `try-catch` block you have to check for errors in the callback
- **no return value:** async functions don't return values, but values will be passed to the callbacks

Let's modify this file a little bit to see how it works in practice:

```
const fs = require('fs')

console.log('start reading a file...')

fs.readFile('file.md', 'utf-8', function (err, content) {
  if (err) {
    console.log('error happened during reading the file')
    return console.log(err)
  }

  console.log(content)
})

console.log('end of the file')
```

The output of this script will be:

```
start reading a file...
end of the file
error happened during reading the file
```

As you can see once we started to read our file the execution continued, and the application printed `end of the file`. Our callback was only called once the file read was finished. How is it possible? **Meet the event loop.**

The Event Loop

The event loop is in the heart of Node.js / Javascript - it is responsible for scheduling asynchronous operations.

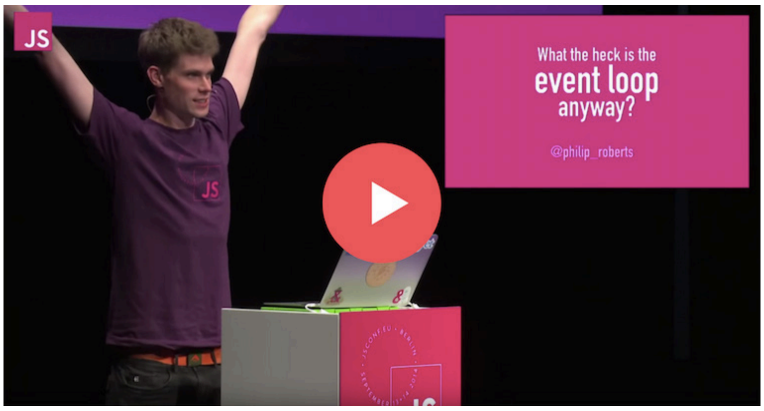
Before diving deeper, let's make sure we understand what event-driven programming is.

> *Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.*

In practice, it means that applications act on events.

Also, as we have already learned in the first chapter, Node.js is single-threaded - from a developer's point of view. It means that you don't have to deal with threads and synchronizing them, Node.js abstracts this complexity away. Everything except your code is executing in parallel.

To understand the event loop more in-depth, continue watching this video:



Async Control Flow

As now you have a basic understanding of how async programming works in JavaScript, let's take a look at a few examples on how you can organize your code.

Async.js

To avoid the so-called Callback-Hell one thing you can do is to start using async.js.

Async.js helps to structure your applications and makes control flow easier.

Let's check a short example of using Async.js, and then rewrite it by using Promises.

The following snippet maps through three files for stats on them:

```
async.parallel(['file1', 'file2', 'file3'], fs.stat,
function (err, results) {
    // results is now an array of stats for each file
})
```

Promises

The Promise object is used for deferred and asynchronous computations. A Promise represents an operation that hasn't completed yet but is expected in the future.

In practice, the previous example could be rewritten as follows:

```
function stats (file) {
  return new Promise((resolve, reject) => {
    fs.stat(file, (err, data) => {
      if (err) {
        return reject (err)
      }
      resolve(data)
    })
  })
}

Promise.all([
  stats('file1'),
  stats('file2'),
  stats('file3')
])
.then((data) => console.log(data))
.catch((err) => console.log(err))
```

Of course, if you use a method that has a Promise interface, then the Promise example can be a lot less in line count as well.

In the next chapter, you will learn how to fire up **your first Node.js HTTP server.**

YOUR FIRST NODE.JS SERVER

In this chapter, I'll guide you how you can fire up a simple Node.js HTTP server and start serving requests.

The `http` module for your Node.js server

When you start building HTTP-based applications in Node.js, the built-in `http/https` modules are the ones you will interact with.

Now, let's create your first Node.js HTTP server! We'll need to require the `http` module and bind our server to the port `3000` to listen on.

```
// content of index.js
const http = require('http')
const port = 3000

const requestHandler = (request, response) => {
  console.log(request.url)
  response.end('Hello Node.js Server!')
}

const server = http.createServer(requestHandler)

server.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err)
  }

  console.log(`server is listening on ${port}`)
})
```

You can start it with:

```
$ node index.js
```

Things to notice here:

- `requestHandler`: **this function will be invoked every time a request hits the server**. If you visit `localhost:3000` from your browser, two log messages will appear: one for / and one for `favicon.ico`
- `if (err)`: error handling - if the port is already taken, or for any other reason our server cannot start, we get notified here

The `http` module is very low-level - creating a complex web application using the snippet above is very time-consuming. This is the reason why we usually pick a framework to work with for our projects. There are a lot you can pick from, but these are the most popular ones:

- express
- hapi
- koa
- restify

For this and the next chapters we are going to use Express, as you will find the most modules on NPM for Express.

Express

> *Fast, unopinionated, minimalist web framework for Node.js - http://expressjs.com/*

Adding Express to your project is only an NPM install away:

```
$ npm install express --save
```

Once you have Express installed, let's see how you can create a similar application as before:

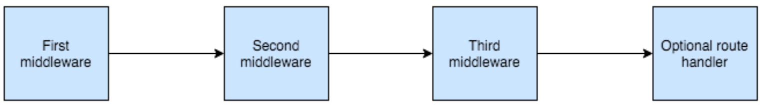
```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (request, response) => {
  response.send('Hello from Express!')
})

app.listen(port, (err) => {
  if (err) {
    return console.log('something bad happened', err)
  }

  console.log(`server is listening on ${port}`)
})
```

The biggest difference what you have to notice here is that Express by default gives you a router. You don't have to check manually for the URL to decide what to do, but instead, you define the application's routing with `app.get`, `app.post`, `app.put`, etc. They are translated to the corresponding HTTP verbs.

One of the most powerful concepts that Express implements is the middleware pattern.

Middlewares

You can think of middlewares as Unix pipelines, but for HTTP requests.



In the diagram you can see how a request can go through an Express application. It travels to three middlewares. Each can modify it, then based on the business logic either the third middleware can send back a response or it can be a route handler.

In practice, you can do it this way:

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

app.use((request, response, next) => {
  console.log(request.headers)
 next()
})

app.use((request, response, next) => {
  request.chance = Math.random()
  next()
})

app.get('/', (request, response) => {
  response.json({
    chance: request.chance
  })
})

app.listen(3000)
```

Things to notice here:

- `app.use`: this is how you can define middlewares - it takes a function with three parameters, the first being the request, the second the response and the third one is the `next` callback. Calling `next` signals Express that it can jump to the next middleware or route handler.
- The first middleware just logs the headers and instantly calls the next one.
- The seconds one adds an extra property to it - **this is one of the most powerful features of the middleware pattern**. Your middlewares can append extra data to the request object that downstream middlewares can read/alter.

Error handling

As in all frameworks, getting the error handling right is crucial. In Express you have to create a special middleware function to do so - a middleware with four parameters:

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

app.get('/', (request, response) => {
  throw new Error('oops')
})

app.use((err, request, response, next) => {
  // log the error, for now just console.log
  console.log(err)
  response.status(500).send('Something broke!')
})
```

Things to notice here:

- The error handler function should be the last function added with `app.use`.
- The error handler has a `next` callback - it can be used to chain multiple error handlers.

Rendering HTML

So far we have taken a look on how to send JSON responses - it is time to learn how to render HTML the easy way. For that, we are going to use the handlebars package with the express-handlebars wrapper.

First, let's create the following directory structure:

```
├── index.js
└── views
    ├── home.hbs
    └── layouts
        └── main.hbs
```

Once you have that, populate index.js with the following snippet:

```
// index.js
const path = require('path')
const express = require('express')
const exphbs = require('express-handlebars')

app.engine('.hbs', exphbs({
  defaultLayout: 'main',
   extname: '.hbs',
  layoutsDir: path.join(__dirname, 'views/layouts')
}))
app.set('view engine', '.hbs')
app.set('views', path.join(__dirname, 'views'))
```

The code above initializes the handlebars engine and sets the layouts directory to `views/layouts`. This is the directory where your layouts will be stored.

Once you have this setup, you can put your initial `html` into the `main.hbs` - to keep things simple let's go with this one:

```
<html>
  <head>
    <title>Express handlebars</title>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

You can notice the `{{{body}}}` placeholder - this is where your content will be placed - let's create the `home.hbs!`

```
<h2>Hello {{name}}<h2>
```

The last thing we have to do to make it work is to add a route handler to our Express application:

```
app.get('/', (request, response) => {
  response.render('home', {
    name: 'John'
  })
})
```

The `render` method takes two parameters:

- The first one is the name of the view,
- and the second is the data you want to render.

Once you call that endpoint you will end up with something like this:

```
<html>
  <head>
    <title>Express handlebars</title>
  </head>
  <body>
    <h2>Hello John<h2>
  </body>
</html>
```

This is just the tip of the iceberg - to learn how to add more layouts and even partials, please refer to the official express-handlebars documentation.

Debugging Express

In some cases, you may need to see what happens with Express when your application is running. To do so, you can pass the following environment variable to Express: `DEBUG=express*`.

You have to start your Node.js HTTP server using:

```
$ DEBUG=express* node index.js
```

Summary

This is how can you set up your first Node.js HTTP server from scratch. I recommend Express to begin with, then feel free to experiment. Let me know how did it go in the comments.

In the next chapter, you will learn **how to retrieve information from databases**.

USING DATABASES

In the following Node.js database chapter, I'll show you how you can set up a Node.js application with a database, and teach you the basics of using it.

Storing data in a global variable

Serving static pages for users - as you have learned it in the previous chapter - can be suitable for landing pages, or for personal blogs. However, if you want to deliver personalized content you have to store the data somewhere.

Let's take a simple example: user signup. You can serve customized content for individual users or make it available for them after identification only.

If a user wants to sign up for your application, you might want to create a route handler to make it possible:

```
const users = []

app.post('/users', function (req, res) {
    // retrieve user posted data from the body
     const user = req.body
    users.push({
      name: user.name,
      age: user.age
    })
    res.send('successfully registered')
})
```

This way you can store the users in a global variable, which will reside in memory for the lifetime of your application.

Using this method might be problematic for several reasons:

- RAM is expensive,
- memory resets each time you restart your application,
- if you don't clean up, sometimes you'll end up with stack overflow.

Storing data in a file

The next thing that might come up in your mind is to store the data in files.

If we store our user data permanently on the file system, we can avoid the previously listed problems.

This method looks like the following in practice:

```
const fs = require('fs')

app.post('/users', function (req, res) {
    const user = req.body
    fs.appendToFile('users.txt', JSON.stringify({ name:
user.name, age: user.age }), (err) => {
        res.send('successfully registered')
    })
})
```

This way we won't lose user data, not even after a server reset. This solution is also cost efficient, since buying storage is cheaper than buying RAM.

Unfortunately storing user data this way still has a couple of flaws:

- Appending is okay, but think about updating or deleting.
- If we're working with files, there is no easy way to access them in parallel (system-wide locks will prevent you from writing).
- When we try to scale our application up, we cannot split files (you can, but it is way beyond the level of this tutorial) in between servers.

This is where real databases come into play.

You might have already heard that there are two main kinds of databases: SQL and NoSQL.

SQL

Let's start with SQL. It is a query language designed to work with relational databases. SQL has a couple of flavors depending on the product you're using, but the fundamentals are same in each of them.

The data itself will be stored in tables, and each inserted piece will be represented as a row in the table, just like in Google Sheets, or Microsoft Excel.

Within an SQL database, you can define schemas - these schemas will provide a skeleton for the data you'll put in there. The types of the different values have to be set before you can store your data. For example, you'll have to define a table for your user data, and have to tell the database that it has a username which is a string, and age, which is an integer type.

NoSQL

On the other hand, NoSQL databases have become quite popular in the last decade. With NoSQL you don't have to define a schema and you can store any arbitrary JSON. This is handy with JavaScript because we can turn any object into a JSON pretty easily. Be careful, because you can never guarantee that the data is consistent, and you can never know what is in the database.

Node.js and MongoDB

There is a common misconception with Node.js what we hear all the time:

"Node.js can only be used with MongoDB (which is the most popular NoSQL database)."

According to my experience, this is not true. There are drivers available for most of the databases, and they also have libraries on NPM. In my opinion, they are as straightforward and easy to use as MongoDB.

Node.js and PostgreSQL

For the sake of simplicity, we're going to use SQL in the following example. My dialect of choice is PostgreSQL.

To have PostgreSQL up and running you have to install it on your computer. If you're on a Mac, you can use homebrew to install PostgreSQL. Otherwise, if you're on Linux, you can install it with your package manager of choice.

For further information read this excellent guide on getting your first database up and running.

If you're planning to use a database browser tool, I'd recommend the command line program called `psql` - it's bundled with the PostgreSQL server installation. Here's a small cheat sheet that will come handy if you start using it.

If you don't like the command-line interface, you can use pgAdmin which is an open source GUI tool for PostgreSQL administration.

Note that SQL is a language on its own, we won't cover all of its features, just the simpler ones. To learn more, there are a lot of great courses online that cover all the basics on PostgreSQL.

Node.js Database Interaction

First, we have to create the database we are going to use. To do so, enter the following command in the terminal: `createdb node_hero`

Then we have to create the table for our users.

```
CREATE TABLE users(
   name VARCHAR(20),
   age SMALLINT
);
```

Finally, we can get back to coding. Here is how you can interact with your database via your Node.js program.

```
 'use strict'

const pg = require('pg')
const conString = 'postgres://username:password@
localhost/node_hero' // make sure to match your own
database's credentials

pg.connect(conString, function (err, client, done) {
  if (err) {
    return console.error('error fetching client from
pool', err)
  }
  client.query('SELECT $1::varchar AS my_first_query',
['node hero'], function (err, result) {
    done()

    if (err) {
      return console.error('error happened during
query', err)
    }
    console.log(result.rows[0])
    process.exit(0)
  })
})
```

This was just a simple example, a 'hello world' in PostgreSQL. Notice that the first parameter is a string which is our SQL command, the second parameter is an array of values that we'd like to parameterize our query with.

It is a huge security error to insert user input into databases as they come in. This protects you from SQL Injection attacks, which is a kind of attack when the attacker tries to exploit severely sanitized SQL queries. Always take this into consideration when building any user facing application. To learn more, check out our Node.js Application Security checklist.

Let's continue with our previous example.

```
app.post('/users', function (req, res, next) {
  const user = req.body

  pg.connect(conString, function (err, client, done) {
    if (err) {
      // pass the error to the express error handler
      return next(err)
    }
    client.query('INSERT INTO users (name, age) VALUES
($1, $2);', [user.name, user.age], function (err,
result) {
      done() //this done callback signals the pg driver
that the connection can be closed or returned to the
connection pool

      if (err) {
        // pass the error to the express error handler
        return next(err)
      }

      res.send(200)
    })
  })
})
```

Achievement unlocked: the user is stored in the database! :) Now let's try retrieving them. Next, let's add a new endpoint to our application for user retrieval.

```
app.get('/users', function (req, res, next) {
  pg.connect(conString, function (err, client, done) {
    if (err) {
      // pass the error to the express error handler
      return next(err)
    }
    client.query('SELECT name, age FROM users;', [],
function (err, result) {
      done()

      if (err) {
         // pass the error to the express error handler
        return next(err)
      }

      res.json(result.rows)
    })
  })
})
```

That wasn’t that hard, was it?

Now you can run any complex SQL query that you can come up with within your Node.js application.

> *With this technique, you can store data persistently in your application, and thanks to the hard-working team of the node-postgres module, it is a piece of cake to do so.*

We have gone through all the basics you have to know about using databases with Node.js. Now go, and create something yourself.

Try things out and experiment, because that’s the best way of becoming a real Node Hero! Practice and be prepared for the next chapter on **how to communicate with third-party APIs**!

THE REQUEST MODULE

In the following chapter, you will learn the basics of HTTP, and how you can fetch resources from external sources using the Node.js request module.

What's HTTP?

HTTP stands for Hypertext Transfer Protocol. HTTP functions as a request–response protocol in the client–server computing model.

HTTP Status Codes

Before diving into the communication with other APIs, let's review the HTTP status codes we may encounter during the process. They describe the outcome of our requests and are essential for error handling.

- 1xx - Informational
- 2xx - Success: These status codes indicate that our request was received and processed correctly. The most common success codes are `200 OK`, `201 Created` and `204 No Content`.
- 3xx - Redirection: This group shows that the client had to do an additional action to complete the request. The most common redirection codes are `301 Moved Permanently`, `304 Not Modified`.
- 4xx - Client Error: This class of status codes is used when the request sent by the client was faulty in some way. The server response usually contains the explanation of the error. The most common client error codes are `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `409 Conflict`.
- 5xx - Server Error: These codes are sent when the server failed to fulfill a valid request due to some error. The cause may be a bug in the code or some temporary or permanent incapability. The most common server error codes are `500 Internal Server Error`, `503 Service Unavailable`. If you'd like to learn more about HTTP status codes, you can find a detailed explanation about them here.

Sending Requests to External APIs

Connecting to external APIs is easy in Node. You can just require the core HTTP module and start sending requests.

Of course, there are much better ways to call an external endpoint. On NPM you can find multiple modules that can make this process easier for you. For example, the two most popular ones are the request and superagent modules.

Both of these modules have an error-first callback interface that can lead to some issues (I bet you've heard about Callback-Hell), but luckily we have access to the promise-wrapped versions.

Using the Node.js Request Module

Using the request-promise module is simple. After installing it from NPM, you just have to require it:

```
const request = require('request-promise')
```

Sending a GET request is as simple as:

```
const options = {
  method: 'GET',
  uri: 'https://risingstack.com'
}

request(options)
  .then(function (response) {
    // Request was successful, use the response object
at will
  })
  .catch(function (err) {
    // Something bad happened, handle the error
  })
```

If you are calling a JSON API, you may want the request-promise to parse the response automatically. In this case, just add this to the request options:

```
json: true
```

POST requests work in a similar way:

```
const options = {
  method: 'POST',
  uri: 'https://risingstack.com/login',
  body: {
    foo: 'bar'
  },
  json: true
    // JSON stringifies the body automatically
}

request(options)
  .then(function (response) {
    // Handle the response
  })
  .catch(function (err) {
    // Deal with the error
  })
```

To add query string parameters you just have to add the `qs` property to the options object:

```
const options = {
  method: 'GET',
  uri: 'https://risingstack.com',
  qs: {
    limit: 10,
    skip: 20,
    sort: 'asc'
  }
}
```

This will make your request URL: `https://risingstack.com?limit=10&skip=20&sort=asc`. You can also define any header the same way we added the query parameters:

```
const options = {
  method: 'GET',
  uri: 'https://risingstack.com',
  headers: {
    'User-Agent': 'Request-Promise',
    'Authorization': 'Basic QWxhZGRpbjpPcGVuU2VzYW1l'
  }
}
```

Error handling

Error handling is an essential part of making requests to external APIs, as we can never be sure what will happen to them. Apart from our client errors the server may respond with an error or just send data in a wrong or inconsistent format. Keep these in mind when you try handling the response. Also, using catch for every request is a good way to avoid the external service crashing our server.

Putting it together

As you have already learned how to spin up a Node.js HTTP server, how to render HTML pages, and how to get data from external APIs, it is time to put them together!

In this example, we are going to create a small Express application that can render the current weather conditions based on city names.

(*To get your AccuWeather API key, please visit their* [*developer site*])

```
const express = require('express')
const rp = require('request-promise')
const exphbs = require('express-handlebars')

const app = express()

app.engine('.hbs', exphbs({
  defaultLayout: 'main',
  extname: '.hbs',
  layoutsDir: path.join(__dirname, 'views/layouts')
}))
app.set('view engine', '.hbs')
app.set('views', path.join(__dirname, 'views'))

app.get('/:city', (req, res) => {
  rp({
    uri: 'http://apidev.accuweather.com/locations/v1/
search',
    qs: {
      q: req.params.city,
      apiKey: 'api-key'
         // Use your accuweather API key here
    },
    json: true
  })
    .then((data) => {
      res.render('index', data)
    })
    .catch((err) => {
      console.log(err)
      res.render('error')
    })
})

app.listen(3000)
```

The example above does the following:

- creates an Express server
- sets up the handlebars structure - for the `.hbs` file please refer to the Node.js HTTP tutorial
- sends a request to the external API
    - ◊ if everything is ok, it renders the page
    - ◊ otherwise, it shows the error page and logs the error

In the next chapter of Node Hero you are going to learn **how to structure your Node.js projects** correctly.

PROJECT STRUCTURING

Most Node.js frameworks don’t come with a fixed directory structure and it might be challenging to get it right from the beginning. In this chapter, you will learn how to properly structure a Node.js project to avoid confusion when your applications start to grow.

The 5 fundamental rules of Project Structuring

There are a lot of possible ways to organize a Node.js project - and each of the known methods has their ups and downs. However, according to our experience, developers always want to achieve the same things: clean code and the possibility of adding new features with ease.

In the past years at RisingStack, we had a chance to build efficient Node applications in many sizes, and we gained numerous insights regarding the dos and donts of project structuring.

We have outlined five simple guiding rules which we enforce during Node.js development. If you manage to follow them, your projects will be fine:

Rule 1:
Organize your Files Around Features, Not Roles

Imagine, that you have the following directory structure:

```
// DON’T
.
├── controllers
|   ├── product.js
|   └── user.js
├── models
|   ├── product.js
|   └── user.js
├── views
|   ├── product.hbs
|   └── user.hbs
```

The problems with this approach are:
- to understand how the product pages work, you have to open up three different directories, with lots of context switching,
- you end up writing long paths when requiring modules: `require(‘../../controllers/user.js’)`

Instead of this, you can structure your Node.js applications around product features / pages / components. It makes understanding a lot easier:

```
// DO
.
├── product
|   ├── index.js
|   ├── product.js
|   └── product.hbs
├── user
|   ├── index.js
|   ├── user.js
|   └── user.hbs
```

Rule 2: Don't Put Logic in `index.js` Files

Use these files only to export functionality, like:

```
// product/index.js
var product = require('./product')

module.exports = {
  create: product.create
}
```

Rule 3: Place Your Test Files Next to The Implementation

Tests are not just for checking whether a module produces the expected output, they also document your modules (*you will learn more on testing in the upcoming chapters*). Because of this, it is easier to understand if test files are placed next to the implementation.

Put your additional test files to a separate `test` folder to avoid confusion.

```
.
├── test
|   └── setup.spec.js
├── product
|   ├── index.js
|   ├── product.js
|   ├── product.spec.js
|   └── product.hbs
├── user
|   ├── index.js
|   ├── user.js
|   ├── user.spec.js
|   └── user.hbs
```

Rule 4: Use a `config` Directory

To place your configuration files, use a `config` directory.

```
.
├── config
|   ├── index.js
|   └── server.js
├── product
|   ├── index.js
|   ├── product.js
|   ├── product.spec.js
|   └── product.hbs
```

Rule 5:
Put Your Long npm Scripts in a `scripts` Directory

Create a separate directory for your additional long scripts in package.json

```
.
├── scripts
|   ├── syncDb.sh
|   └── provision.sh
├── product
|   ├── index.js
|   ├── product.js
|   ├── product.spec.js
|   └── product.hbs
```

In the next chapter of Node Hero, you are going to learn how to **authenticate users using Passport.js.**

NODE.JS AUTHENTICATION USING PASSPORT.JS

In this chapter, you are going to learn how to implement a local Node.js authentication strategy using Passport.js and Redis.

Technologies to use

Before jumping into the actual coding, let's take a look at the new technologies we are going to use in this chapter.

What is Passport.js?
Simple, unobtrusive authentication for Node.js - passportjs.org

Passport is authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.

```
passport.authenticate('github');
```

Passport is an authentication middleware for Node.js which we are going to use for session management.

What is Redis?

Redis is an open source (BSD licensed), in-memory data structure store, used as database, cache and message broker. - redis.io

We are going to store our user's session information in Redis, and not in the process's memory. This way our application will be a lot easier to scale.

The Demo Application

For demonstration purposes, let's build an application that does only the following:

- exposes a login form,
- exposes two protected pages:
  - ◊ a profile page,
  - ◊ secured notes

The Project Structure

You have already learned how to structure Node.js projects in the previous chapter of Node Hero, so let's use that knowledge!

We are going to use the following structure:

```
├── app
|   ├── authentication
|   ├── note
|   ├── user
|   ├── index.js
|   └── layout.hbs
├── config
|   └── index.js
├── index.js
└── package.json
```

As you can see we will organize files and directories around features. We will have a user page, a note page, and some authentication related functionality.

(Download the full source code at https://github.com/RisingStack/nodehero-authentication)

The Node.js Authentication Flow

Our goal is to implement the following authentication flow into our application:

1. User enters username and password
2. The application checks if they are matching
3. If they are matching, it sends a `Set-Cookie` header that will be used to authenticate further pages
4. When the user visits pages from the same domain, the previously set cookie will be added to all the requests
5. Authenticate restricted pages with this cookie

To set up an authentication strategy like this, follow these three steps:

1. Set up Express
2. Set up Passport for Node.js
3. Add Protected Endpoints

Step 1: Setting up Express

We are going to use Express for the server framework - you can learn more on the topic by reading our Express tutorial.

```
// file:app/index.js
const express = require('express')
const passport = require('passport')
const session = require('express-session')
const RedisStore = require('connect-redis')(session)

const app = express()
app.use(session({
  store: new RedisStore({
    url: config.redisStore.url
  }),
  secret: config.redisStore.secret,
  resave: false,
  saveUninitialized: false
}))
app.use(passport.initialize())
app.use(passport.session())
```

What did we do here?

First of all, we required all the dependencies that the session management needs. After that we have created a new instance from the `express-session` module, which will store our sessions.

For the backing store, we are using Redis, but you can use any other, like MySQL or MongoDB.

Step 2: Setting up Passport for Node.js

Passport is a great example of a library using plugins. For this tutorial, we are adding the `passport-local` module which enables easy integration of a simple local authentication strategy using usernames and passwords.

For the sake of simplicity, in this example (see the next page), we are not using a second backing store, but only an in-memory user instance. In real life applications, the `findUser` would look up a user in a database.

```
// file:app/authenticate/init.js
const passport = require('passport')
const LocalStrategy = require('passport-local').Strategy

const user = {
  username: 'test-user',
  password: 'test-password',
  id: 1
}

passport.use(new LocalStrategy(
  function(username, password, done) {
    findUser(username, function (err, user) {
      if (err) {
        return done(err)
      }
      if (!user) {
        return done(null, false)
      }
      if (password !== user.password  ) {
        return done(null, false)
      }
      return done(null, user)
    })
  }
))
```

Once the `findUser` returns with our user object the only thing left is to compare the user-fed and the real password to see if there is a match.

If it is a match, we let the user in (by returning the user to passport - `return done(null, user)`), if not we return an unauthorized error (by returning nothing to passport - `return done(null)`).

Step 3: Adding Protected Endpoints

To add protected endpoints, we are leveraging the middleware pattern Express uses. For that, let's create the authentication middleware first:

```
// file:app/user/init.js
const passport = require('passport')

app.get('/profile', passport.authenticationMiddleware(),
renderProfile)
```

It only has one role if the user is authenticated (has the right cookies) it simply calls the next middleware; otherwise it redirects to the page where the user can log in.

Using it is as easy as adding a new middleware to the route definition.

```
// file:app/authentication/middleware.js
function authenticationMiddleware () {
  return function (req, res, next) {
    if (req.isAuthenticated()) {
      return next()
    }
    res.redirect('/')
  }
}
```

Summary

In this Node.js tutorial, you have learned how to add basic authentication to your application. Later on, you can extend it with different authentication strategies, like Facebook or Twitter. You can find more strategies at http://passportjs.org/.

The full, working example is on GitHub, you can take a look here: https://github.com/RisingStack/nodehero-authentication

---

The next chapter of Node Hero will be all about **unit testing Node.js applications.** You will learn concepts like unit testing, test pyramid, test doubles and a lot more!

UNIT TESTING

In this chapter, you are going to learn what is unit testing in Node.js, and how to test your applications properly.

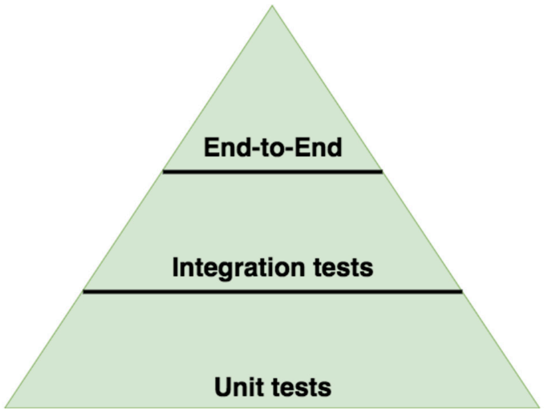
Testing Node.js Applications

You can think of tests as safeguards for the applications you are building. They will run not just on your local machine, but also on the CI services so that failing builds won't get pushed to production systems.

"Tests are more than just safeguards - they provide a living documentation for your codebase."

You may ask: *what should I test in my application? How many tests should I have?*

The answer varies across use-cases, but as a rule of thumb, you can follow the guidelines set by the **test pyramid.**



Essentially, the test pyramid describes that you should write **unit tests, integration tests and end-to-end tests** as well. You should have more integration tests than end-to-end tests, and even more unit tests.

Let's take a look at how you can add unit tests for your applications!

Please note, that we are not going to talk about integration tests and end-to-end tests here as they are far beyond the scope of this tutorial.

Unit Testing Node.js Applications

We write unit tests to see if a given module (unit) works. All the dependencies are stubbed, meaning we are providing fake dependencies for a module.

You should write the test for the exposed methods, not for the internal workings of the given module.

The Anatomy of a Unit Test

Each unit test has the following structure:

1. Test setup
2. Calling the tested method
3. Asserting

Each unit test should test one concern only. *(Of course this doesn't mean that you can add one assertion only).*

Modules Used for Node.js Unit Testing

For unit testing, we are going to use the following modules:

- **test runner:** mocha, alternatively tape
- **assertion library:** chai, alternatively the **assert** module *(for asserting)*
- **test spies, stubs and mocks:** sinon *(for test setup)*.

Spies, stubs and mocks - which one and when?

Before doing some hands-on unit testing, let's take a look at what spies, stubs and mocks are!

Spies

You can use spies to get information on function calls, like how many times they were called, or what arguments were passed to them.

```
it('calls subscribers on publish', function () {
  var callback = sinon.spy()
  PubSub.subscribe('message', callback)

  PubSub.publishSync('message')

  assertTrue(callback.called)
})
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
```

Stubs

Stubs are like spies, but they replace the target function. You can use stubs to control a method’s behaviour to force a code path (like throwing errors) or to prevent calls to external resources (like HTTP APIs).

```
it(‘calls all subscribers, even if there are
exceptions’, function (){
  var message = ‘an example message’
  var error = ‘an example error message’
  var stub = sinon.stub().throws()
  var spy1 = sinon.spy()
  var spy2 = sinon.spy()

  PubSub.subscribe(message, stub)
  PubSub.subscribe(message, spy1)
  PubSub.subscribe(message, spy2)

  PubSub.publishSync(message, undefined)

  assert(spy1.called)
  assert(spy2.called)
  assert(stub.calledBefore(spy1))
})
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
```

Mocks

A mock is a fake method with a pre-programmed behavior and expectations.

```
 it(‘calls all subscribers when exceptions happen’,
function () {
  var myAPI = {
    method: function () {}
  }

  var spy = sinon.spy()
  var mock = sinon.mock(myAPI)
  mock.expects(“method”).once().throws()

  PubSub.subscribe(“message”, myAPI.method)
  PubSub.subscribe(“message”, spy)
  PubSub.publishSync(“message”, undefined)

  mock.verify()
  assert(spy.calledOnce)
// example taken from the sinon documentation site:
http://sinonjs.org/docs/
})
```

As you can see, for mocks you have to define the expectations upfront.

Imagine, that you'd like to test the following module:

```
const fs = require('fs')
const request = require('request')

function saveWebpage (url, filePath) {
  return getWebpage(url, filePath)
    .then(writeFile)
}v

function getWebpage (url) {
  return new Promise (function (resolve, reject) {
    request.get(url, function (err, response, body) {
      if (err) {
        return reject(err)
      }

      resolve(body)
    })
  })
}

function writeFile (fileContent) {
  let filePath = 'page'
  return new Promise (function (resolve, reject) {
    fs.writeFile(filePath, fileContent, function (err) {
      if (err) {
        return reject(err)
      }

      resolve(filePath)
    })
  })
}

module.exports = {
  saveWebpage
}
```

This module does one thing: it saves a web page (based on the given URL) to a file on the local machine. To test this module we have to stub out both the fs module as well as the `request` module.

Before actually starting to write the unit tests for this module, at RisingStack, we usually add a `test-setup.spec.js` file to do basics test setup, like creating sinon sandboxes. This saves you from writing `sinon.sandbox.create()` and `sinon.sandbox.restore()` after each tests.

```
// test-setup.spec.js
const sinon = require('sinon')
const chai = require('chai')

beforeEach(function () {
  this.sandbox = sinon.sandbox.create()
})

afterEach(function () {
  this.sandbox.restore()
})
```

Also, please note, that we always put test files next to the implementation, hence the `.spec.js name`. In our `package.json` you can find these lines:

```
{
  "test-unit": "NODE_ENV=test mocha '/**/*.spec.js'",
}
```

Once we have these setups, it is time to write the tests itself!

```
const fs = require('fs')
const request = require('request')

const expect = require('chai').expect

const webpage = require('./webpage')

describe('The webpage module', function () {
  it('saves the content', function * () {
    const url = 'google.com'
    const content = '<h1>title</h1>'
    const writeFileStub = this.sandbox.stub(fs,
'writeFile', function (filePath, fileContent, cb) {
      cb(null)
    })

    const requestStub = this.sandbox.stub(request,
'get', function (url, cb) {
      cb(null, null, content)
    })

    const result = yield webpage.saveWebpage(url)

    expect(writeFileStub).to.be.calledWith()
    expect(requestStub).to.be.calledWith(url)
    expect(result).to.eql('page')
  })
})
```

The full codebase can be found here: https://github.com/RisingStack/nodehero-testing

Code coverage

To get a better idea of how well your codebase is covered with tests, you can generate a coverage report.

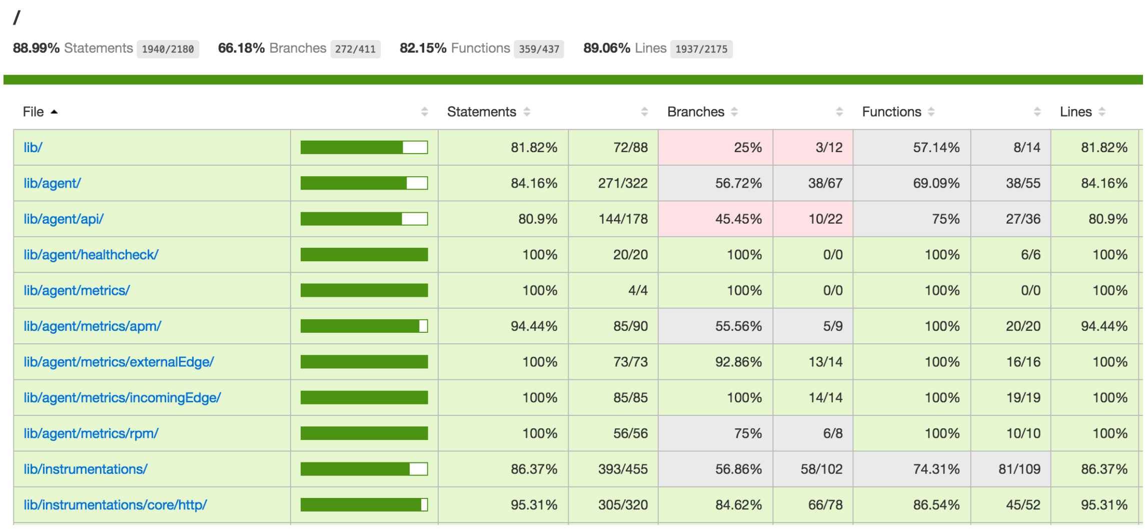
This report will include metrics on:

- **line** coverage,
- **statement** coverage,
- **branch** coverage,
- and **function** coverage.

At RisingStack, we use istanbul for code coverage. You should add the following script to your `package.json` to use `istanbul` with `mocha`:

```
 istanbul cover _mocha $(find ./lib -name \"*.spec.js\"
-not -path \"./node_modules/*\")
```

Once you do, you will get something like this:



/

88.99% Statements 1940/2180 **66.18%** Branches 272/411 **82.15%** Functions 359/437 **89.06%** Lines 1937/2175

File ▲		Statements		Branches		Functions		Lines
lib/		81.82%	72/88	25%	3/12	57.14%	8/14	81.82%
lib/agent/		84.16%	271/322	56.72%	38/67	69.09%	38/55	84.16%
lib/agent/api/		80.9%	144/178	45.45%	10/22	75%	27/36	80.9%
lib/agent/healthcheck/		100%	20/20	100%	0/0	100%	6/6	100%
lib/agent/metrics/		100%	4/4	100%	0/0	100%	0/0	100%
lib/agent/metrics/apm/		94.44%	85/90	55.56%	5/9	100%	20/20	94.44%
lib/agent/metrics/externalEdge/		100%	73/73	92.86%	13/14	100%	16/16	100%
lib/agent/metrics/incomingEdge/		100%	85/85	100%	14/14	100%	19/19	100%
lib/agent/metrics/rpm/		100%	56/56	75%	6/8	100%	10/10	100%
lib/instrumentations/		86.37%	393/455	56.86%	58/102	74.31%	81/109	86.37%
lib/instrumentations/core/http/		95.31%	305/320	84.62%	66/78	86.54%	45/52	95.31%

You can click around, and actually see your source code annotated - which part is tested, which part is not.

Testing can save you a lot of trouble - still, it is inevitable to also do debugging from time to time. In the next chapter of Node Hero, you are going to learn how to **debug Node.js applications**.

DEBUGGING

In this chapter, you are going to learn debugging your Node.js applications using the debug module, the built-in Node debugger and Chrome's developer tools.

Bugs, debugging

The term **bug** and **debugging** have been a part of engineering jargon for many decades. One of the first written mentions of bugs is as follows:

> *It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that "Bugs" — as such little faults and difficulties are called— show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.*
> ***Thomas Edison***

Debugging Node.js Applications

One of the most frequently used approach to find issues in Node.js applications is the heavy usage of `console.log` for debugging.

> *"Console.log is efficient for debugging small snippets but we recommend better alternatives!"*

Let's take a look at them!

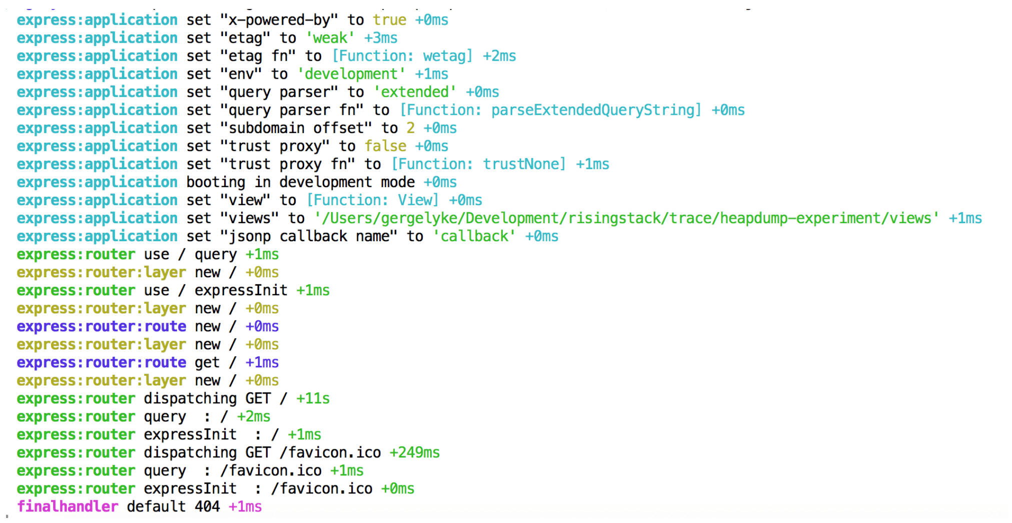
The `debug` module

Some of the most popular modules that you can `require` into your project come with the `debug` module. With this module, you can enable third-party modules to log to the standard output, `stdout`. To check whether a module is using it, take a look at the `package.json` file's dependency section.

To use the `debug` module, you have to set the `DEBUG` environment variable when starting your applications. You can also use the `*` character to wildcard names. The following line will print all the `express` related logs to the standard output.

```
DEBUG=express* node app.js
```

The output will look like this:



```
express:application set "x-powered-by" to true +0ms
express:application set "etag" to 'weak' +3ms
express:application set "etag fn" to [Function: wetag] +2ms
express:application set "env" to 'development' +1ms
express:application set "query parser" to 'extended' +0ms
express:application set "query parser fn" to [Function: parseExtendedQueryString] +0ms
express:application set "subdomain offset" to 2 +0ms
express:application set "trust proxy" to false +0ms
express:application set "trust proxy fn" to [Function: trustNone] +1ms
express:application booting in development mode +0ms
express:application set "view" to [Function: View] +0ms
express:application set "views" to '/Users/gergelyke/Development/risingstack/trace/heapdump-experiment/views' +1ms
express:application set "jsonp callback name" to 'callback' +0ms
express:router use / query +1ms
express:router:layer new / +0ms
express:router use / expressInit +1ms
express:router:layer new / +0ms
express:router:route new / +0ms
express:router:layer new / +0ms
express:router:route get / +1ms
express:router:layer new / +0ms
express:router dispatching GET / +11s
express:router query  : / +2ms
express:router expressInit  : / +1ms
express:router dispatching GET /favicon.ico +249ms
express:router query  : /favicon.ico +1ms
express:router expressInit  : /favicon.ico +0ms
finalhandler default 404 +1ms
```

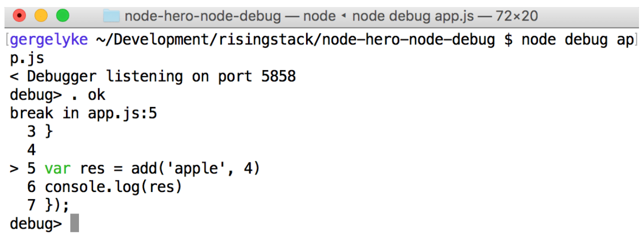
The Built-in Node.js Debugger

> *Node.js includes a full-featured out-of-process debugging utility accessible via a simple TCP-based protocol and built-in debugging client.*

To start the built-in debugger you have to start your application this way:

```
node debug app.js
```

Once you have done that, you will see something like this:



```
gergelyke ~/Development/risingstack/node-hero-node-debug $ node debug ap
p.js
< Debugger listening on port 5858
debug> . ok
break in app.js:5
  3 }
  4
> 5 var res = add('apple', 4)
  6 console.log(res)
  7 });
debug>
```

Basic Usage of the Node Debugger

To navigate this interface, you can use the following commands:

- `c` => continue with code execution
- `n` => execute this line and go to next line
- `s` => step into this function
- `o` => finish function execution and step out
- `repl` => allows code to be evaluated remotely

You can add breakpoints to your applications by inserting the `debugger` statement into your codebase.

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

var res = add('apple', 4)
```

Watchers

> *It is possible to watch expression and variable values during debugging. On every breakpoint, each expression from the watchers list will be evaluated in the current context and displayed immediately before the breakpoint's source code listing.*

To start using watchers, you have to define them for the expressions you want to watch. To do so, you have to do it this way:

```
watch('expression')
```

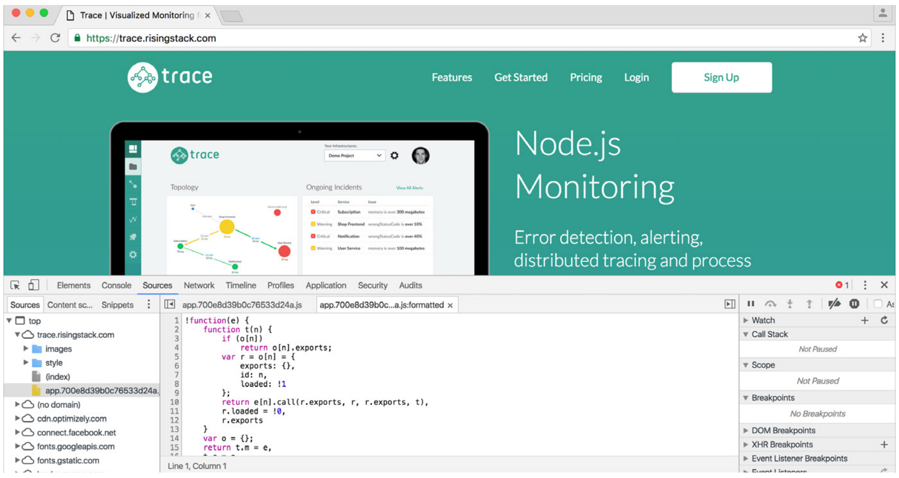
To get a list of active watchers type `watchers`, to unwatch an expression use `unwatch('expression')`.

> *Pro tip: you can switch running Node.js processes into debug mode by sending the SIGUSR1 command to them. After that you can connect the debugger with node debug -p <pid>.*

To understand the full capabilities of the built-in debugger, check out the official API docs: https://nodejs.org/api/debugger.html.

The Chrome Debugger

When you start debugging complex applications, something visual can help. Wouldn't be great to use the familiar UI of the Chrome DevTools for debugging Node.js applications as well?



```
!function(e) {
    function t(n) {
        if (o[n])
            return o[n].exports;
        var r = o[n] = {
            exports: {},
            id: n,
            loaded: !1
        };
        return e[n].call(r.exports, r, r.exports, t),
        r.loaded = !0,
        r.exports
    }
    var o = {};
    return t.m = e,
```

Good news, the Chrome debug protocol is already ported into a Node.js module and can be used to debug Node.js applications.

To start using it, you have to install `node-inspector` first:

```
npm install -g node-inspector
```

Once you installed it, you can start debugging your applications by starting them this way:

```
node-debug index.js --debug-brk
```

(the `--debug-brk` *pauses the execution on the first line)*

It will open up the Chrome Developer tools and you can start to debug your Node.js applications with it.

In the next chapter of Node Hero, you are going to learn how to secure your Node.js applications.

SECURITY

In this Node.js security chapter, you are going to learn how to defend your applications against the most common attack vectors.

Node.js Security threats

Nowadays we see almost every week some serious security breaches, like in the LinkedIn or MySpace cases. During these attacks, a huge amount of user data was leaked - as well as corporate reputations damaged.

Studies also show that security related bug tickets are open for an average of 18 months in some industries.

We have to fix this attitude. **If you develop software, security is a part of your job.**

Start the Node.js Security Tutorial

Let's get started, and secure our Node.js application by proper coding, tooling, and operation!

Secure Coding Style

Rule 1: Don't use `eval`

Eval can open up your application for code injection attacks. Try not to use it, but if you have to, never inject unvalidated user input into `eval`.

Eval is not the only one you should avoid - in the background each one of the following expressions uses eval:
- `setInterval(String, 2)`
- `setTimeout(String, 2)`
- `new Function(String)`

Rule 2: Always use strict mode

With `'use strict'` you can opt in to use a restricted "variant" of JavaScript. It eliminates some silent errors and will throw them all the time.

```
'use strict'
delete Object.prototype
// TypeError
var obj = {
    a: 1,
    a: 2
}
// syntax error
```

Rule 3: Handle errors carefully

During different error scenarios, your application may leak sensitive details about the underlying infrastructure, like: `X-Powered-By:Express`.

Stack traces are not treated as vulnerabilities by themselves, but they often reveal information that can be interesting to an attacker. Providing debugging information as a result of operations that generate errors is considered a bad practice. You should always log them, but never show them to the users.

Rule 4: Do a static analysis of your codebase

Static analysis of your application's codebase can catch a lot of errors. For that we suggest using ESLint with the Standard code style.

Running Your Services in Production Securely

Using proper code style is not enough to efficiently secure Node.js applications - you should also be careful about how you run your services in production.

Rule 5: Don't run your processes with superuser rights

Sadly, we see this a lot: developers are running their Node.js application with superuser rights, as they want it to listen on port 80 or 443.

This is just wrong. In the case of an error/bug, your process can bring down the entire system, as it has credentials to do anything.

Instead of this, what you can do is to set up an HTTP server/proxy to forward the requests. This can be nginx or Apache. Check out our article on Operating Node.js in Production to learn more.

Rule 6: Set up the obligatory HTTP headers

There are some security-related HTTP headers that your site should set. These headers are:

- **Strict-Transport-Security** enforces secure (HTTP over SSL/TLS) connections to the server
- **X-Frame-Options** provides clickjacking protection
- **X-XSS-Protection** enables the Cross-site scripting (XSS) filter built into most recent web browsers
- **X-Content-Type-Options** prevents browsers from MIME-sniffing a response away from the declared content-type
- **Content-Security-Policy** prevents a wide range of attacks, including Cross-site scripting and other cross-site injections

In Node.js it is easy to set these using the Helmet module:

```
var express = require('express')
var helmet = require('helmet')

var app = express()

app.use(helmet())
```

Helmet is available for Koa as well: koa-helmet.

Rule 7: Do proper session management

The following list of flags should be set for each cookie:

- **secure** - this attribute tells the browser to only send the cookie if the request is being sent over HTTPS.
- **HttpOnly** - this attribute is used to help prevent attacks such as cross-site scripting since it does not allow the cookie to be accessed via JavaScript.

Rule 8: Set cookie scope

- **domain** - this attribute is used to compare against the domain of the server in which the URL is being requested. If the domain matches or if it is a sub-domain, then the path attribute will be checked next.
- **path** - in addition to the domain, the URL path that the cookie is valid for can be specified. If the domain and path match, then the cookie will be sent in the request.
- **expires** - this attribute is used to set persistent cookies since the cookie does not expire until the set date is exceeded.

In Node.js you can easily create this cookie using the cookies package. Again, this is quite low-level, so you will probably end up using a wrapper, like the cookie-session.

```
var cookieSession = require('cookie-session')
var express = require('express')

var app = express()

app.use(cookieSession({
  name: 'session',
  keys: [
    process.env.COOKIE_KEY1,
    process.env.COOKIE_KEY2
  ]
}))

app.use(function (req, res, next) {
  var n = req.session.views || 0
  req.session.views = n++
  res.end(n + ' views')
})

app.listen(3000)
```

(The example is taken from the cookie-session module documentation.)

Tools to Use

Congrats, you're almost there! If you followed this tutorial and did the previous steps thoroughly, you have just one area left to cover regarding Node.js security. Let's dive into using the proper tools to look for module vulnerabilities!

Rule 9: Look for vulnerabilities with Retire.js

> *"Always look for vulnerabilities in your nodejs modules. You are what you require."*

The goal of Retire.js is to help you detect the use of module versions with known vulnerabilities.

Simply install with:

```
npm install -g retire
```

After that, running it with the retire command is going to look for vulnerabilities in your `node_modules` directory. (Also note, that retire.js works not only with node modules but with front end libraries as well.)

Rule 10: Audit your modules with the Node Security Platform CLI

`nsp` is the main command line interface to the Node Security Platform. It allows for auditing a `package.json` or `npm-shrinkwrap.json` file against the NSP API to check for vulnerable modules.

```
npm install nsp --global
# From inside your project directory
nsp check
```

Node.js security is not a big deal after all is it? I hope you found these rules to be helpful for securing your Node.js applications - and will follow them in the future since security is a part of your job!

If you'd like to read more on Node.js security, I can recommend these articles to start with:

- Node.js Security Tips
- OWASP's Top Ten Cheat Sheet
- Node.js security checklist

In the next chapter of Node Hero, you are going to learn how to deploy your secured Node.js application, so people can actually start using it!

DEPLOYING YOUR APPLICATION

In this chapter about Node.js deployment, you are going to learn how to deploy Node.js applications to either a PaaS provider (Heroku) or with using Docker.

Deploy Node.js to a PaaS

> *Platform-as-a-Service providers can be a great fit for teams who want to do zero operations or create small applications.*

In this part of the tutorial, you are going to learn how to use Heroku to deploy your Node.js applications with ease.

> *"Heroku can be a great fit for teams who want to do zero ops or create small apps"*

Prerequisites for Heroku

To deploy to Heroku, we have to push code to a remote git repository. To achieve this, add your public key to Heroku. After registration, head over to your account and save it there *(alternatively, you can do it with the CLI)*.

We will also need to download and install the Heroku toolbelt. To verify that your installation was successful, run the following command in your terminal:

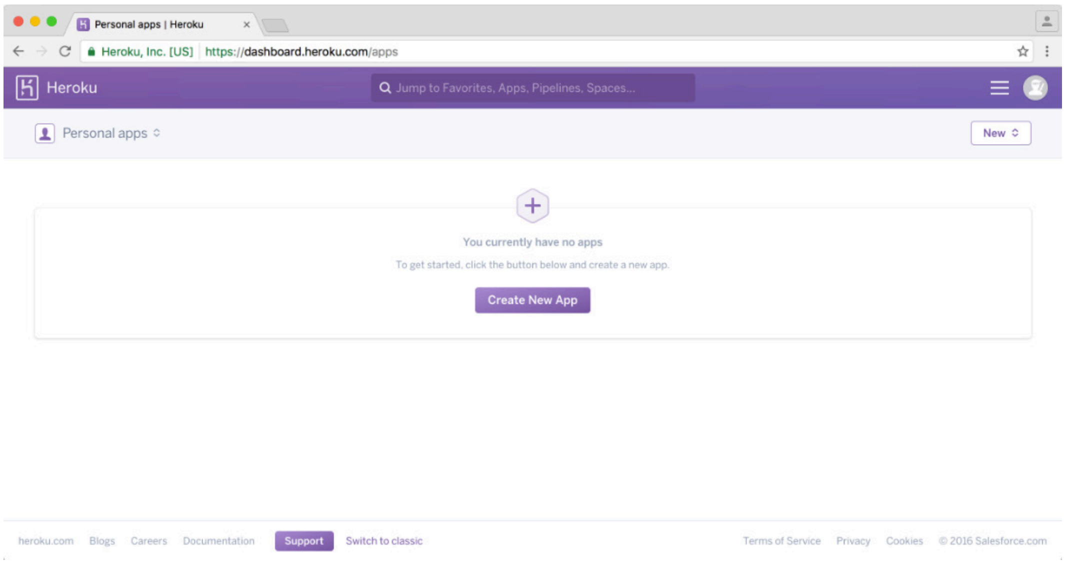
```
heroku --version
heroku-toolbelt/3.40.11 (x86_64-darwin10.8.0) ruby/1.9.3
```

Once the toolbelt is up and running, log in to use it:

```
heroku login
Enter your Heroku credentials.
Email: joe@example.com
Password:
```

(For more information on the toolkit, head over to the Heroku Devcenter)

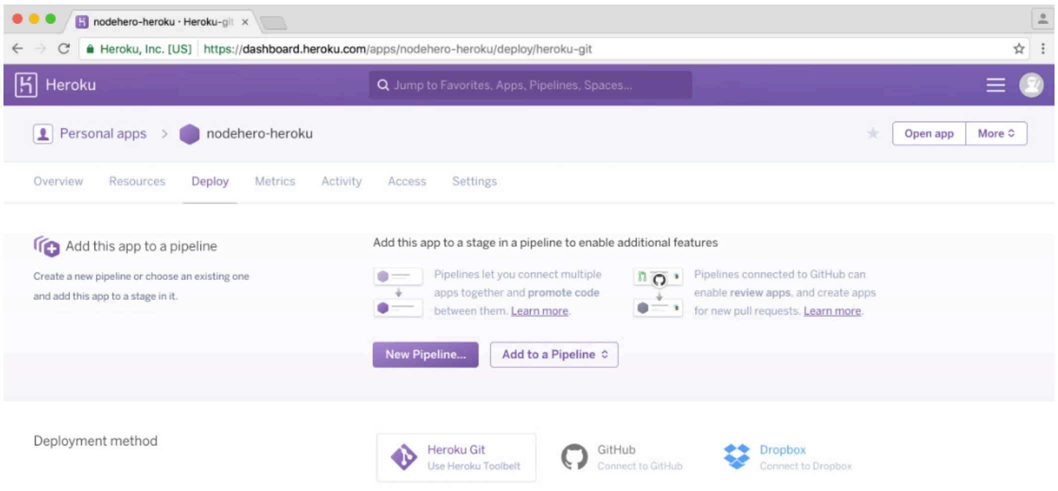
Deploying to Heroku


Personal apps | Heroku
Heroku, Inc. [US] https://dashboard.heroku.com/apps
Heroku
Jump to Favorites, Apps, Pipelines, Spaces...
Personal apps
New
Create New App
heroku.com Blogs Careers Documentation Support Switch to classic
Terms of Service Privacy Cookies © 2016 Salesforce.com

You currently have no apps

To get started, click the button below and create a new app.

Click **Create New App**, add a new and select a region. In a matter of seconds, your application will be ready, and the following screen will welcome you:


nodehero-heroku · Heroku-git
Heroku, Inc. [US] https://dashboard.heroku.com/apps/nodehero-heroku/deploy/heroku-git
Heroku
Jump to Favorites, Apps, Pipelines, Spaces...
Personal apps > nodehero-heroku
Open app
More
Overview Resources Deploy Metrics Activity Access Settings
Add this app to a pipeline
New Pipeline...
Add to a Pipeline
Deployment method
Heroku Git
Use Heroku Toolbelt
GitHub
Connect to GitHub
Dropbox
Connect to Dropbox

Create a new pipeline or choose an existing one and add this app to a stage in it.

Add this app to a stage in a pipeline to enable additional features

Pipelines let you connect multiple apps together and promote code between them. Learn more.

Pipelines connected to GitHub can enable review apps, and create apps for new pull requests. Learn more.

Go to the **Settings** page of the application, and grab the Git URL. In your terminal, add the Heroku remote url:

```
git remote add heroku HEROKU_URL
```

You are ready to deploy your first application to Heroku - it is really just a git push away:

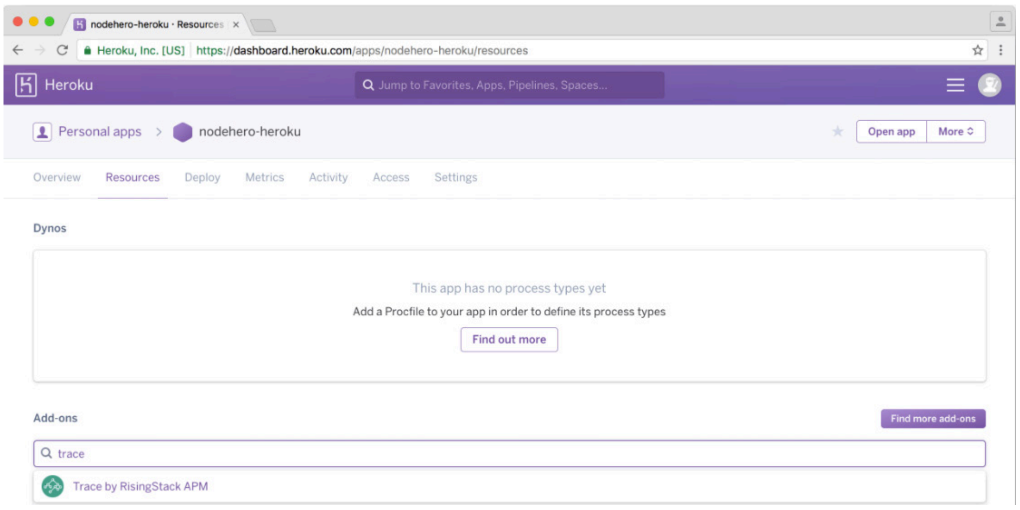
```
git push heroku master
```

Once you do this, Heroku starts building your application and deploys it as well. After the deployment, your service will be reachable at https://YOUR-APP-NAME.herokuapp.com.

Heroku Add-ons

One of the most valuable part of Heroku is its ecosystem since there are dozens of partners providing databases, monitoring tools, and other solutions.

To try out an add-on, install Trace, our Node.js monitoring solution. To do so, look for **Add-ons** on your application's page, and start typing Trace, then click on it to provision. Easy, right?


nodehero-heroku · Resources
Heroku, Inc. [US] https://dashboard.heroku.com/apps/nodehero-heroku/resources
Heroku
Jump to Favorites, Apps, Pipelines, Spaces...
Personal apps
nodehero-heroku
Open app
More
Overview
Resources
Deploy
Metrics
Activity
Access
Settings
Dynos
This app has no process types yet
Add a Procfile to your app in order to define its process types
Find out more
Add-ons
Find more add-ons
trace
Trace by RisingStack APM

(To finish the Trace integration, follow our Heroku guide.)

Deploy Node.js using Docker

> *In the past years Docker gained a massive momentum and became the go-to containerization software. In this part of the tutorial, you are going to learn how to create images from your Node.js applications and run them.*

"Docker for Node.js is a great choice if you want more control and save on costs"

In this part of the tutorial, you are going to learn how to create images from your Node.js applications and run them.

Docker Basics

To get started with Docker, download and install it from the Docker website.

Putting a Node.js application inside Docker

First, we have to get two definitions right:

- **Dockerfile:** you can think of the Dockerfile as a receipt - it includes instructions on how to create a Docker image
- **Docker image:** the output of the Dockerfile run - this is the runnable unit

In order to run an application inside Docker, we have to write the Dockerfile first.

Dockerfile for Node.js

In the root folder of your project, create a `Dockerfile`, an empty text file, then paste the following code into it:

```
FROM risingstack/alpine:3.3-v4.2.6-1.1.3

COPY package.json package.json
RUN npm install

# Add your source files
COPY . .
CMD [“npm”,”start”]
```

Things to notice here:

- `FROM`: describes the base image used to create a new image - in this case it is from the public Docker Hub
- `COPY`: this command copies the `package.json` file to the Docker image so that we can run `npm install` inside
- `RUN`: this runs commands, in this case `npm install`
- `COPY` again - note, that we have done the copies in two separate steps. The reason is, that Docker creates layers from the command results, so if our `package.json` is not changing, it won’t do `npm install` again
- `CMD`: a Docker image can only have one `CMD` - this defines what process should be started with the image

Once you have the `Dockerfile`, you can create an image from it using:

```
docker build .
```

> *Using private NPM modules? Check out our tutorial on how to install private NPM modules in Docker!*

After the successful build of your image, you can list them with:

```
docker images
```

To run an image:

```
docker run IMAGE_ID
```

Congratulations! You have just run a Dockerized Node.js application locally. Time to deploy it!

Deploying Docker Images

One of the great things about Docker is that once you have a build image, you can run it everywhere - most environments will just simply `docker pull` your image, and run it.

Some providers that you can try:

- AWS BeanStalk
- Heroku Docker Support
- Docker Cloud
- Kubernetes on Google Cloud - (I highly recommend to read our article on moving to Kubernetes from our PaaS provider)

Setting them up is very straightforward - if you run into any problems, feel free to ask in the comments section!

In the next chapter of Node Hero, you are going to **learn how to monitor your Node.js applications** - so that it can be online 24/7.

MONITORING NODE.JS APPLICATIONS

In the last chapter of the series, I'm going to show you how to do Node.js monitoring and how to find advanced issues in production environments.

The Importance of Node.js Monitoring

Getting insights into production systems is critical when you are building Node.js applications! You have an obligation to constantly detect bottlenecks and figure out what slows your product down.

An even greater issue is to handle and preempt downtimes. You must be notified as soon as they happen, preferably before your customers start to complain. Based on these needs, proper monitoring should give you at least the following features and insights into your application's behavior:

- **Profiling on a code level:** You have to understand how much time does it take to run each function in a production environment, not just locally.
- **Monitoring network connections:** If you are building a microservices architecture, you have to monitor network connections and lower delays in the communication between your services.
- **Performance dashboard:** Knowing and constantly seeing the most important performance metrics of your application is essential to have a fast, stable production system.
- **Real-time alerting:** For obvious reasons, if anything goes down, you need to get notified immediately. This means that you need tools that can integrate with Pagerduty or Opsgenie - so your DevOps team won't miss anything important.

"Getting insights into production systems is critical when you are building nodejs applications"

Server Monitoring versus Application Monitoring

One concept developers usually apt to confuse is monitoring servers and monitoring the applications themselves. As we tend to do a lot of virtualization, these concepts should be treated separately, as a single server can host dozens of applications.
Let's go trough the major differences!

Server Monitoring

Server monitoring is responsible for the host machine. It should be able to help you answer the following questions:

- Does my server have enough disk space?
- Does it have enough CPU time?
- Does my server have enough memory?
- Can it reach the network?

For server monitoring, you can use tools like zabbix.

Application Monitoring

Application monitoring, on the other hand, is responsible for the health of a given application instance. It should let you know the answers to the following questions:

- Can an instance reach the database?
- How much request does it handle?
- What are the response times for the individual instances?
- Can my application serve requests? Is it up?

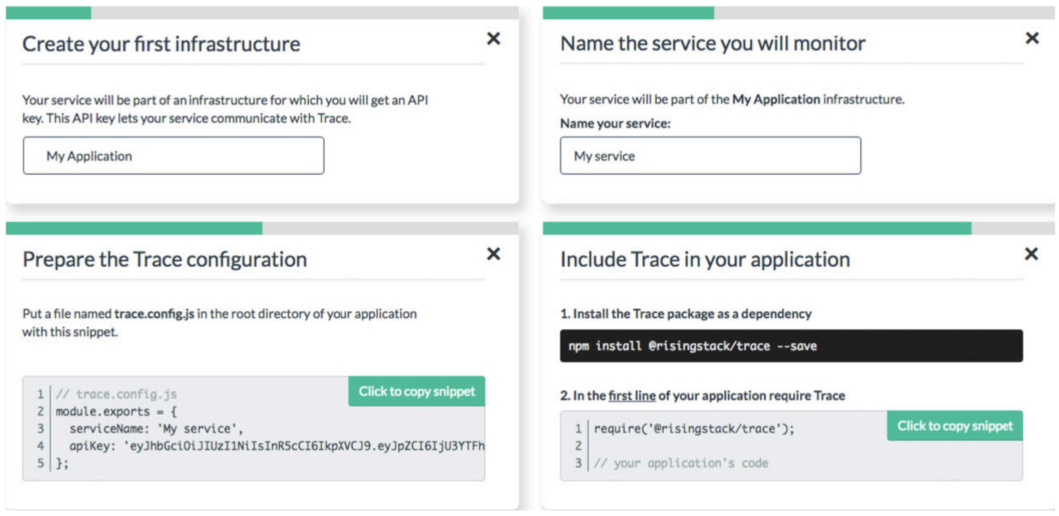
For application monitoring, I recommend using our tool called Trace. What else? :)

We developed it to be an easy to use and efficient tool that you can use to monitor and debug applications from the moment you start building them, up to the point when you have a huge production app with hundreds of services.

How to Use Trace for Node.js Monitoring

To get started with Trace, head over to https://trace.risingstack.com and create your free account!

Once you registered, follow these steps to add Trace to your Node.js applications. It only takes up a minute - and these are the steps you should perform:



Create your first infrastructure

Your service will be part of an infrastructure for which you will get an API key. This API key lets your service communicate with Trace.

My Application

Name the service you will monitor

Your service will be part of the **My Application** infrastructure.

Name your service:

My service

Prepare the Trace configuration

Put a file named **trace.config.js** in the root directory of your application with this snippet.

```
// trace.config.js
module.exports = {
  serviceName: 'My service',
  apiKey: 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjU3YTFh
};
```

Include Trace in your application

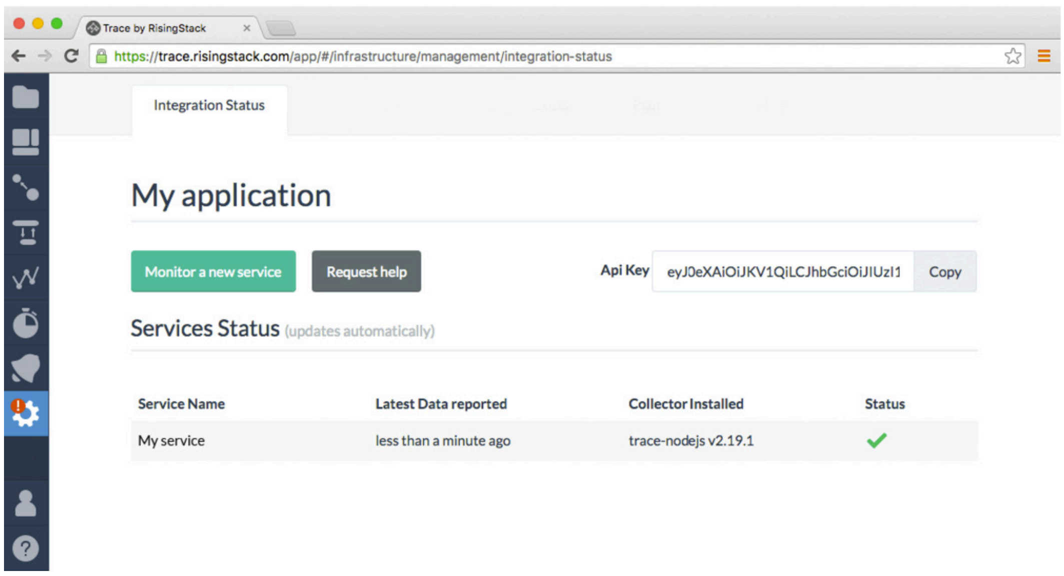
1. Install the Trace package as a dependency

```
npm install @risingstack/trace --save
```

2. In the first line of your application require Trace

```
require('@risingstack/trace');

// your application's code
```

Easy, right? If everything went well, you should see that the service you connected has just started sending data to Trace:


Trace by RisingStack
https://trace.risingstack.com/app/#/infrastructure/management/integration-status
Integration Status
Monitor a new service
Request help
Copy

My application

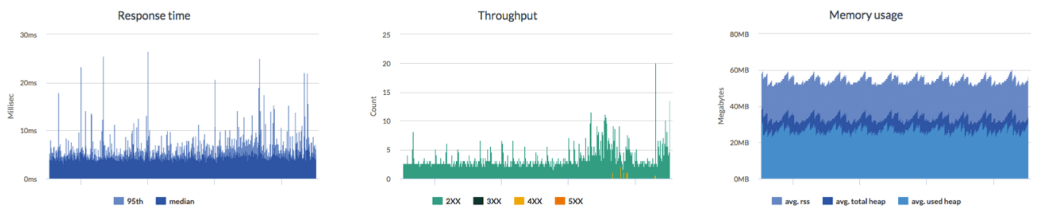
Api Key eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1

Services Status (updates automatically)

Service Name	Latest Data reported	Collector Installed	Status
My service	less than a minute ago	trace-nodejs v2.19.1	✓

#1: Measure your performance

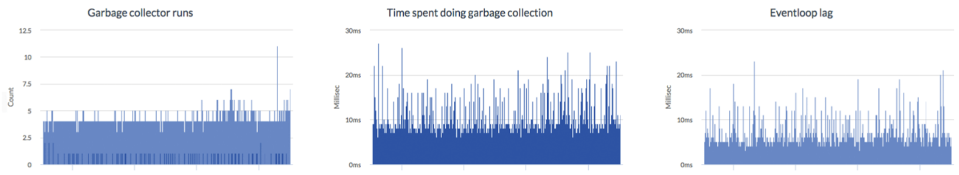
As the first step of monitoring your Node.js application, I recommend to head over to the metrics page and check out the performance of your services.



- You can use the response time panel to check out median and 95th percentile response data. It helps you to figure out when and why your application slows down and how it affects your users.

- The throughput graph shows request per minutes (rpm) for status code categories (200-299 // 300-399 // 400-499 // >500). This way you can easily separate healthy and problematic HTTP requests within your application.

- The memory usage graph shows how much memory your process uses. It's quite useful for recognizing memory leaks and preempting crashes.



If you'd like to see special Node.js metrics, check out the garbage collection and event loop graphs. Those can help you to hunt down memory leaks. Read our metrics documentation.

#2: Set up alerts

As I mentioned earlier, you need a proper alerting system in action for your production application.

Go the alerting page of Trace and click on **Create a new alert.**

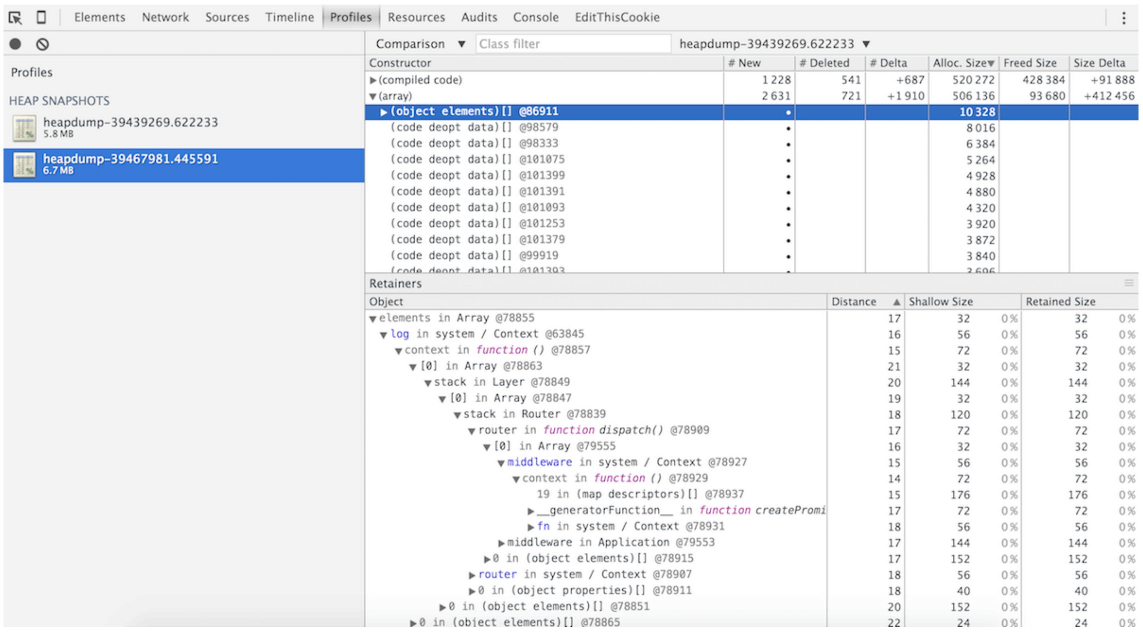
- The most important thing to do here is to set up downtime and memory alerts. Trace will notify you on email / Slack / Pagerduty / Opsgenie, and you can use Webhooks as well.

- I recommend setting up the alert we call Error rate by status code to know about HTTP requests with 4XX or 5XX status codes. These are errors you should definitely care about.

- It can also be useful to create an alert for Response time - and get notified when your app starts to slow down.

#3: Investigate memory heapdumps

Go to the *Profiler* page and request a new memory heapdump, wait 5 minutes and request another. Download them and open them on Chrome DevTool's *Profiles* page. Select the second one (the most recent one), and click *Comparison*.



Constructor	# New	# Deleted	# Delta	Alloc. Size▼	Freed Size	Size Delta
▶(compiled code)	1 228	541	+687	520 272	428 384	+91 888
▼(array)	2 631	721	+1 910	506 136	93 680	+412 456
▶(object elements)[] @86911	•			10 328		
(code deopt data)[] @98579	•			8 016		
(code deopt data)[] @98333	•			6 384		
(code deopt data)[] @101075	•			5 264		
(code deopt data)[] @101399	•			4 928		
(code deopt data)[] @101391	•			4 880		
(code deopt data)[] @101093	•			4 320		
(code deopt data)[] @101253	•			3 920		
(code deopt data)[] @101379	•			3 872		
(code deopt data)[] @99919	•			3 840		
(code deopt data)[] @101393	•			3 696		

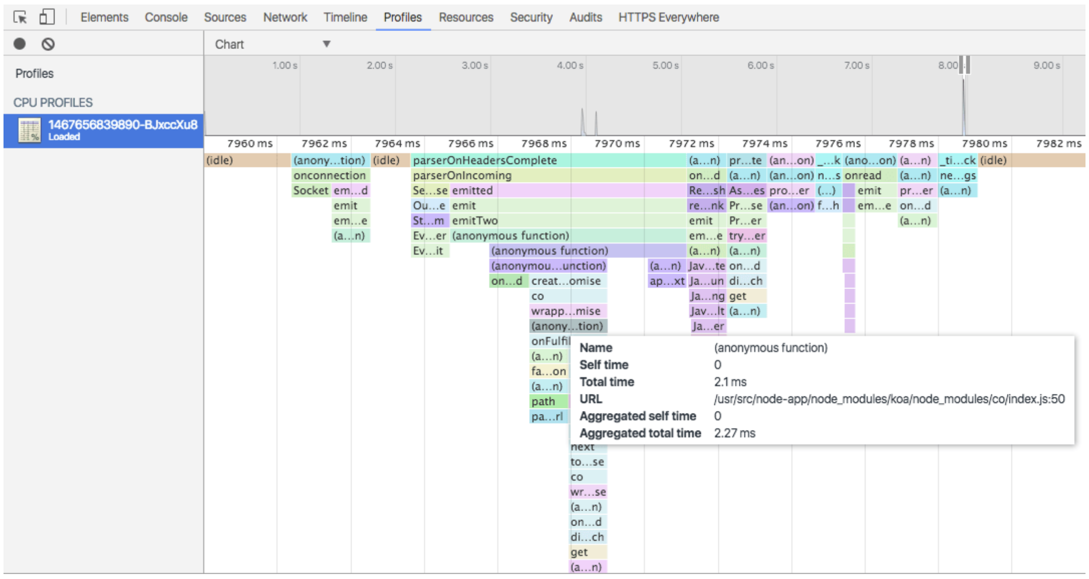
Object	Distance ▲	Shallow Size		Retained Size	
▼elements in Array @78855	17	32	0 %	32	0 %
▼log in system / Context @63845	16	56	0 %	56	0 %
▼context in *function* () @78857	15	72	0 %	72	0 %
▼[0] in Array @78863	21	32	0 %	32	0 %
▼stack in Layer @78849	20	144	0 %	144	0 %
▼[0] in Array @78847	19	32	0 %	32	0 %
▼stack in Router @78839	18	120	0 %	120	0 %
▼router in *function dispatch()* @78909	17	72	0 %	72	0 %
▼[0] in Array @79555	16	32	0 %	32	0 %
▼middleware in system / Context @78927	15	56	0 %	56	0 %
▼context in *function* () @78929	14	72	0 %	72	0 %
19 in (map descriptors)[] @78937	15	176	0 %	176	0 %
▶__generatorFunction__ in *function createPromi*	17	72	0 %	72	0 %
▶fn in system / Context @78931	18	56	0 %	56	0 %
▶middleware in Application @79553	17	144	0 %	144	0 %
▶0 in (object elements)[] @78915	17	152	0 %	152	0 %
▶router in system / Context @78907	18	56	0 %	56	0 %
▶0 in (object properties)[] @78911	18	40	0 %	40	0 %
▶0 in (object elements)[] @78851	20	152	0 %	152	0 %
▶0 in (object elements)[] @78865	22	24	0 %	24	0 %

With this view, you can easily find memory leaks in your application. In a previous article we've written about this process in a detailed way, you can read it here: [Hunting a Ghost - Finding a Memory Leak in Node.js]

#4: CPU profiling

Profiling on the code level is essential to understand how much time does your function take to run in the actual production environment. Luckily, Trace has this area covered too.

All you have to do is to head over to the **CPU Profiles** tab on the Profiling page. Here you can request and download a profile which you can load into the Chrome DevTool as well.Once you loaded it, you'll be able to see the 10 second timeframe of your application and see all of your functions with times and URL's as well.



Name (anonymous function)
Self time 0
Total time 2.1 ms
URL /usr/src/node-app/node_modules/koa/node_modules/co/index.js:50
Aggregated self time 0
Aggregated total time 2.27 ms

Once you loaded it, you'll be able to see the 10 second timeframe of your application and see all of your functions with times and URL's as well.

With this data, you'll be able to figure out what slows down your application and deal with it!

The End

This is it.

During the 13 episodes of the Node Hero series, you learned the basics of building great applications with Node.js.

I hope you enjoyed it and improved a lot! Please share this series with your friends if you think they need it as well - and show them Trace too. It's a great tool for Node.js development!

In case you'd like to continue improving your Node.js skills, we recommend to check out our new series called **Node.js at Scale!**