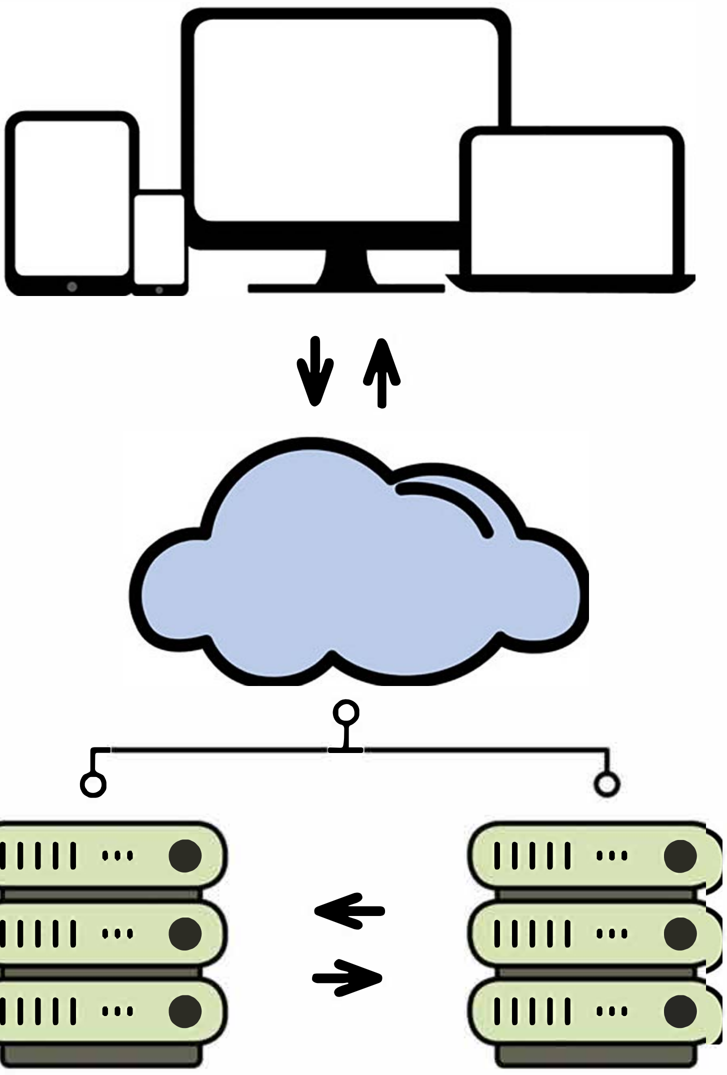
Building APIs with **Node.JS**



Caio Ribeiro Pereira

Building APIs with Node.js

Caio Ribeiro Pereira

Tweet This Book!

Please help Caio Ribeiro Pereira by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the ebook: Building Amazing APIs with Node.js

The suggested hashtag for this book is #NodejsAPI.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#NodejsAPI

Thanks God for everything he has done in my life so far! I'd like to also thank my parents for all the love, the strength, the incentive and for all the support they have gave me since the beginning of my life.

Thanks to the NodeBR community. Their feedback helped me to improve this book. Also, thanks to all the readers of the blog Underground WebDev, after all, the essence of this book is based on many of my posts.

Finally, thank you, dear reader by purchasing this book.

Contents

Preface . . . . **i**
- The current web application scene i
- Who is this book for? i
- How should I study it? i

Introduction to Node.js **1**
- What's Node.js? 1
- Main features 1
- Why do I need to learn Node.js? 2

Setting up the environment **4**
- Node.js standard instalation 4
- Node.js instalation via NVM 5

Managing modules with NPM **7**
- What does NPM do? 7
- Top NPM commands 8
- Understanding package.json file 8
- NPM Task automation 10

Building an API **12**
- Introduction to Express 12
- Getting started the pilot project 13
- Implementing a simple and static resource 18
- Arranging the loading of modules 20

Working with SQL databases **25**
- Introduction to SQLite3 and Sequelize 25
- Setting up Sequelize 27
- Creating models 29

CRUDify API resources **36**
- Organizing tasks routes 36
- Listing tasks via GET 38
- Creating tasks via POST 39

Finding a task via GET 40
Updating a task via PUT 40
Deleting a task via DELETE 41
Refactoring some middlewares 42
Creating user's endpoints 44
Testing endpoint's access using Postman 45

Authenticating users 50
Introduction to Passport.js and JWT 50
Installing Passport and JWT 52
Implementing JWT authentication 53
Generating tokens for authenticated users 55

Testing the application - Part 1 62
Introduction to Mocha 62
Setting up the test environment 62
Testing the authentication endpoint 68

Testing the application - Part 2 72
Testing task's endpoints 72
Testing user's endpoints 78

Documenting the API 84
Introduction to ApiDoc.js 84
Documenting token generation 88
Documenting user's resource 89
Documenting task's resource 92

Preparing the production environment 97
Introduction to CORS 97
Enabling CORS in the API 97
Generating logs 99
Configuring parallel processing using cluster module 102
Compacting requests using GZIP middleware 105
Installing SSL support to use HTTPS 107
Armoring the API with Helmet 108

Building the client-side app - Part 1 111
Setting up the app environment 111
Creating signin and signup views 116
Writing signin and signup components 118

Building the client-side app - Part 2 129
Views and components for task's CRUD 129
Views and components for logged users 134

Creating the main menu 136
Treating all screen's events 138

Preface

The current web application scene

Currently, we live in a period in which the majority of users use different types of devices to connect to the internet. The most popular devices are smartphones, tablets and notebooks. Developing systems for different types of devices requires the work of building web services, also known by the name of API (*Application Program Interface*).

Basically, the APIs are back-end systems that are designed to work only with data in a centralized manner, allowing client-side applications to be developed separately, to have a unique interfaces to the final user. These client-side applications are typically mobile apps, desktop applications or web apps.

Since 2010 Node.js increasingly proved to be an excellent platform to solve many problems, especially for building REST APIs. The **single-thread architecture** that performs **non-blocking I/O** running on top of JavaScript - which is a very present language in all current browsers - showed good performance in the processing of many kind of applications. There are some cases of large companies, such as LinkedIn and PayPal, which significantly saved expenses with servers migrating some of their projects to Node.js.

In addition, another advantage of using the Node.js, which captivated many developers, is the low learning curve. After all, who has worked with web development already have at least a basic understanding of the JavaScript language.

Who is this book for?

This book is targeted for developers who have at least the basic knowledge of JavaScript and, especially, know well about **Object Oriented Programming** (OOP), a little bit of **client-server architecture** and that have an understanding about the main characteristics of **REST APIs**.

To master these concepts, even basic knowledge of them, is essential to fully understand the reading of this book.

Something really cool about this book is that all the codes are written using the latest JavaScript's implementation, the **EcmaScript 2015** (aka **ECMAScript 6** or **ES6**).

How should I study it?

Throughout the reading, many concepts and codes are going to be presented for you to learn in practice all the theoretical part of this book. It will guide you in a didactic way in the development of two projects (an API and a web client application), which in the end both are going to be integrated to work as a single project.

Warning: It is highly recommended to follow step-by-step these book's instructions, so in the end you can conclude the project correctly.

Introduction to Node.js

What's Node.js?

Node.js is a low level and a highly scalable platform. Using Node.js you will program directly with many network and internet protocols or use libraries that have access to OS resources. To program in Node.js you only need to master JavaScript language – that's right, only JavaScript! And the JavaScript runtime used in this platform is the famous Javascript V8, which is also used in Google Chrome.

Main features

Single-Thread

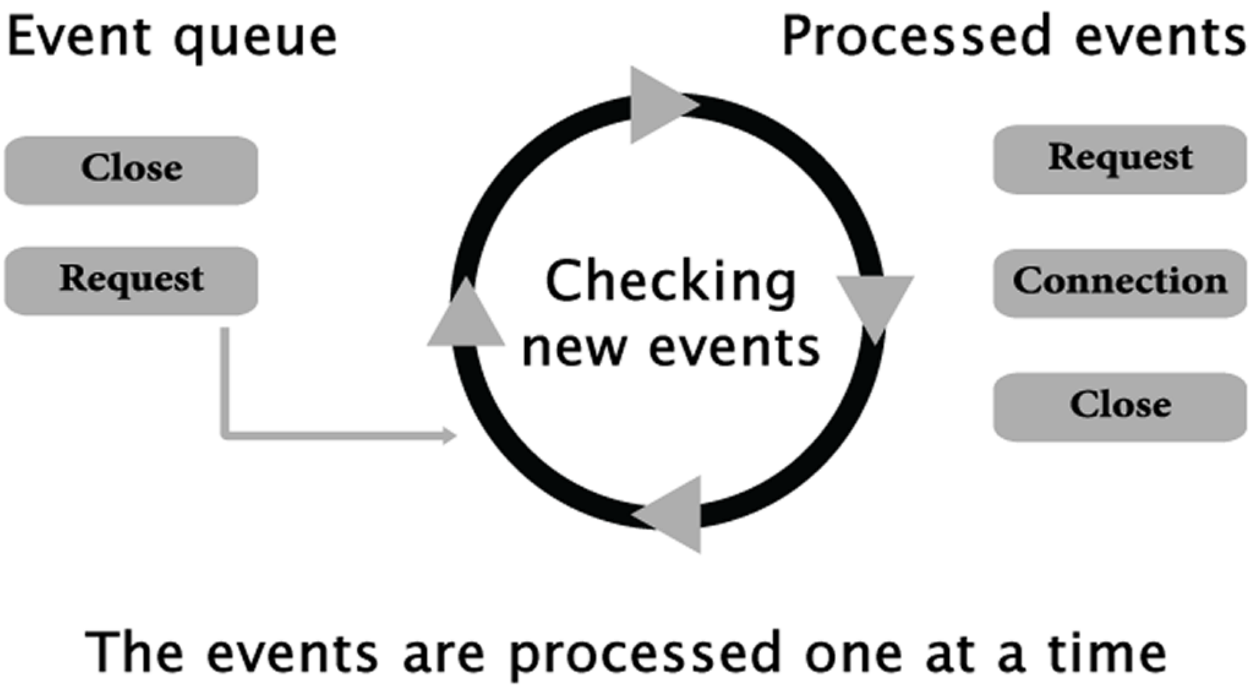
A single-thread architecture works with only the main thread instance for each initiated process. If you are used to programming in multi-thread architecture, like Java or .NET to use the concurrency processing, unfortunately it won't be possible with Node.js. However, you can works with parallel processing using multiple process.

For example, you can use a native library from Node.js called `clusters` that allows you to implement a network of multiple processes. In Node.js you can create **N-1 processes** to work with parallel processing, while the main process (aka master process) works on balancing the load among the other processes (aka slave process). If your server has multiple cores, applying this technique will optimize the processing's usage.

Another way is to use asynchronous programming, which is one of the main resources from JavaScript. The asynchronous functions on Node.js work as **Non-blocking I/O**, that is, in case your application has to read a huge file it will not block the CPU, allowing your application to continue to process other tasks requested by other users.

Event-Loop

Node.js has **event-oriented paradigm**. It follows the same philosophy of event-oriented on JavaScript client-side, the only difference is the type of event, meaning that there are not events such as mouse's `click`, `key up` from the keyboard or any other events from web browsers. In fact, Node.js works with **I/O events**, for example: `connect` from a database, `open` a file or read `data` from a streaming and many others.



The event-loop process

The Event-Loop is the responsible agent in noticing and emitting events. Simply speaking, it is an **infinite loop** which in each iteration, it verifies in the event queue if some event was triggered. When an event is triggered, the Event-Loop execute it and send it to the queue of executed events. When an event is running, we can write any business logic on it inside the callback function commonly used in JavaScript.

Why do I need to learn Node.js?

- **JavaScript everywhere**: Node.js uses JavaScript as a server-side programming language. This feature allows you to cut short your learning process, after all, the language is the same as JavaScript client-side. Your main challenge in this platform is to learn how to works with the asynchronous programming to take most of this technique. Another advantage of working with JavaScript is that you can keep an easy maintenance project – of course, as long as you know who to write codes using JavaScript. You are going to easily find professionals for your projects and you're going to spend less time studying a new server-side language. A technical JavaScript advantage in comparison with other back-end languages is that you will no longer use specifical frameworks to parse JSON objects (JavaScript Object Notation); after all, the JSON from the client-side will be same as in the server-side. There are also cases of applications running databases that persists JSON objects, such as the popular NoSQL: MongoDB, CouchDB and Riak. An important detail is that the Node.js uses many functionalities from ECMAScript 6, allowing you write an elegant and robust JavaScript code.

- **Active community**: This is one of the biggest advantages in Node.js. Currently, there are many communities around the world striving to make Node.js a huge platform, either writing posts and tutorials, giving speeches in some events or publishing and maintaining new modules.
- **Ready to realtime**: Node.js have become popular thanks to some frameworks for realtime communications between client and server. The **SockJS** and **Socket.IO** are good examples. They are compatible with the protocol **WebSockets** and they allow you to traffic data through a single bidirectional connection, treating all data by JavaScript events.
- **Big players**: LinkedIn, Wallmart, Groupon, Microsoft, Netflix, Uber and Paypal are some of the big companies using Node.js and there are many more.

Conclusion

Up to this point, all the theory was explained: main concepts and advantages about why use Node.js. In the next chapters, we are going to see how it works on practice, with a single condition! Open your mind to the new and read this book with excitement to enjoy this book!

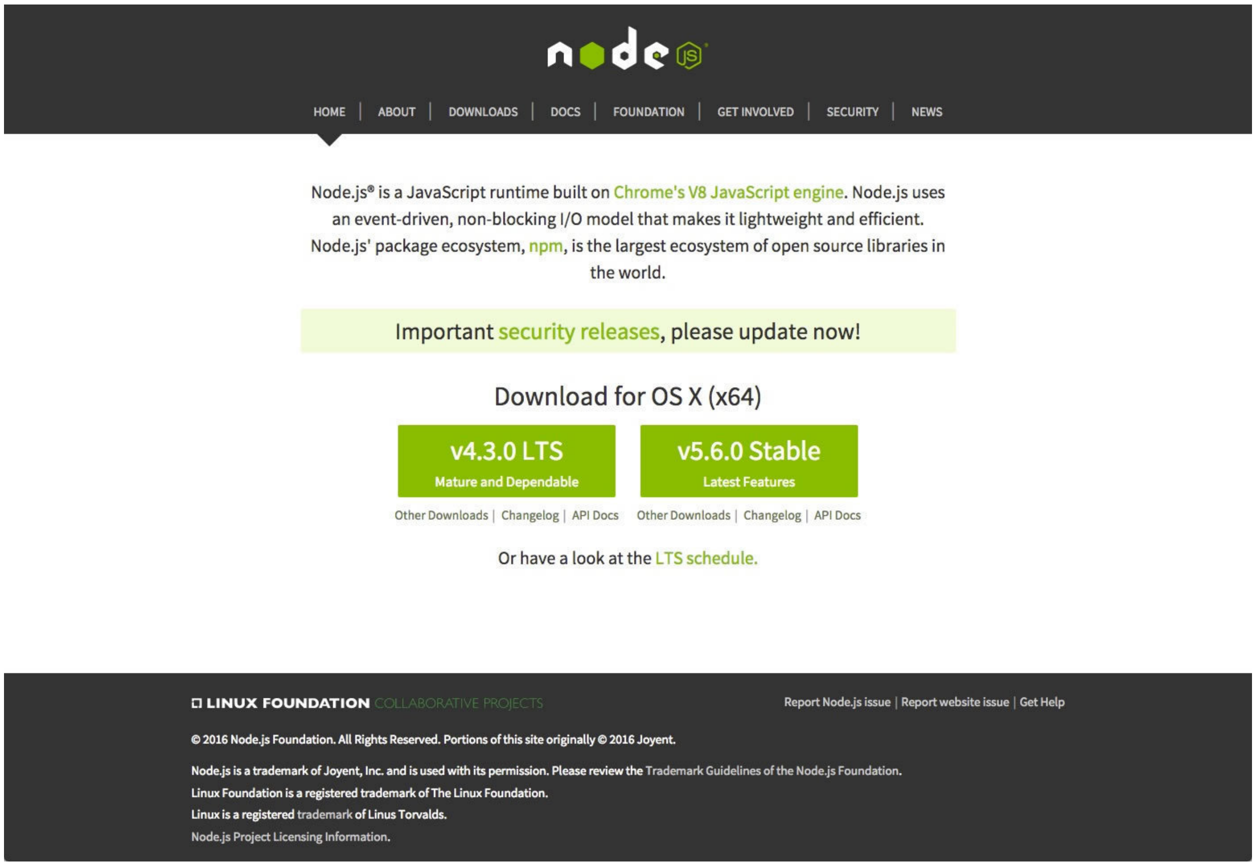
Setting up the environment

In this chapter, we are going to install Node.js on the most used OS (Windows, Linux and MacOSX). However, the entire book will use MacOSX as default OS.

Don't worry about the differences among these operation systems, because all examples in this book are compatible with the all the three OS platforms.

Node.js standard instalation

To set up the Node.js environment is quite the same despite the OS. Just few procedures will be different in each OS, especially for Linux, but it is not a big deal.


HOME | ABOUT | DOWNLOADS | DOCS | FOUNDATION | GET INVOLVED | SECURITY | NEWS

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. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Important security releases, please update now!

Download for OS X (x64)

v4.3.0 LTS
Mature and Dependable
Other Downloads | Changelog | API Docs

v5.6.0 Stable
Latest Features
Other Downloads | Changelog | API Docs

Or have a look at the LTS schedule.

LINUX FOUNDATION COLLABORATIVE PROJECTS

Report Node.js issue | Report website issue | Get Help

© 2016 Node.js Foundation. All Rights Reserved. Portions of this site originally © 2016 Joyent.

Node.js is a trademark of Joyent, Inc. and is used with its permission. Please review the Trademark Guidelines of the Node.js Foundation.
Linux Foundation is a registered trademark of The Linux Foundation.
Linux is a registered trademark of Linus Torvalds.
Node.js Project Licensing Information.

Node.js homepage

The first step is to access the official Node.js website: nodejs.org[1]

[1]http://nodejs.org

Then, click on the button **v5.6.0 Stable** to download the latest and compatible version for your Windows or MacOSX. If you use Linux, I recommend you to read this quick wiki:

https://github.com/nodejs/node/wiki/Installing-and-Building-Node.js

This wiki explains how to compile and install Node.js in some popular Linux distro.

After the download, install it. If you use Windows or MacOSX, just click in some next, next and next buttons until the installation is completed, after all there is none specific configuration to do it.

To test if everything is running correctly, just open the **Terminal** (for Linux or MacOSX), **Command Prompt** or **Power Shell** (for Windows) type this command:

```
node -v && npm -v
```

This will check the version of Node.js and NPM. This book uses **Node v5.6.0** and **NPM 3.6.0**.

About io.js and Node.js merge

Since **September 8th 2015**, Node.js was upgraded from **0.12.x** to **4.0.0** version. This happened because of the merge from **io.js**. The io.js was a fork built and maintained by the Node.js community. They worked to include some features from **ECMAScript 6** implementation, besides making several improvements that were slowly on Node.js. The io.js reached the **3.0.0** version and both groups decided to merge io.js into Node.js, the result was the Node.js **v4.0.0**. This merge not only gave a huge upgrade to this platform, but also gave a more stable and reliable version to Node.js.

Note: This book will use the **Node v5.6.0**, with a lot of ES6 codes. To learn more about ECMAScript 6, take a look this website: http://es6-features.org

Node.js instalation via NVM

Warning: This section, we are just going to explore an alternative way to setup the Node.js using a version manager. Feel free to skip this step in case you don't need or don't want to learn how to use the NVM.

Just like Ruby language has the RVM (*Ruby Version Manager*) to manage multiple Ruby versions, Node.js also has a manager too, the NVM (*Node Version Manager*).

NVM is a perfect solution to test your projects in different Node.js versions. It is also useful for people that like to test unstable or latest versions.

NVM has great advantages, it is practical, easy to use, it can uninstall a Node.js version with a single command and will save you time to search and install some Node.js version. It is a good alternative, especially on Linux which its native package managers are frequently outdated and invalidates a new Node.js version to be installed.

Setup NVM

In few steps you can set up NVM into a MacOSX or Linux system, you just need to run this command:

```
curl https://raw.githubusercontent.com/creationix/nvm/v0.30.2/install.sh | bash
```

Unfortunately, the official version of NVM is not available to Windows, but there are alternative projects created by the community. Here are two similar alternatives for Windows:

- **NVM-Windows**: https://github.com/coreybutler/nvm-windows
- **Nodist**: https://github.com/marcelklehr/nodist

Both have a similar CLI (*Command Line Interface*) inspired by NVM.

Top NVM commands

As a cake recipe, right bellow I show you a list with the top commands that will be essential to manage multiple versions or at least keep your environment updated with the latest version.

- `nvm ls`: lists all versions installed;
- `nvm ls-remote`: lists all the available Node.js versions to download and install;
- `nvm install vX.X.X`: download and install a version;
- `nvm uninstall vX.X.X`: Uninstall an existing version;
- `nvm use vX.X.X`: choose an existing version to use;
- `nvm alias default vX.X.X`: choose an existing version to be loaded by default in the OS boot time;

Attention: change the `vX.X.X` for a Node.js version of your choice, such as `v5.6.0`

Installing Node.js via NVM

To install Node.js via NVM, you just need to run the following commands:

```
nvm install v5.6.0
nvm use v5.6.0
nvm alias default v5.6.0
```

After running these commands, you will have the Node.js installed and ready to use.

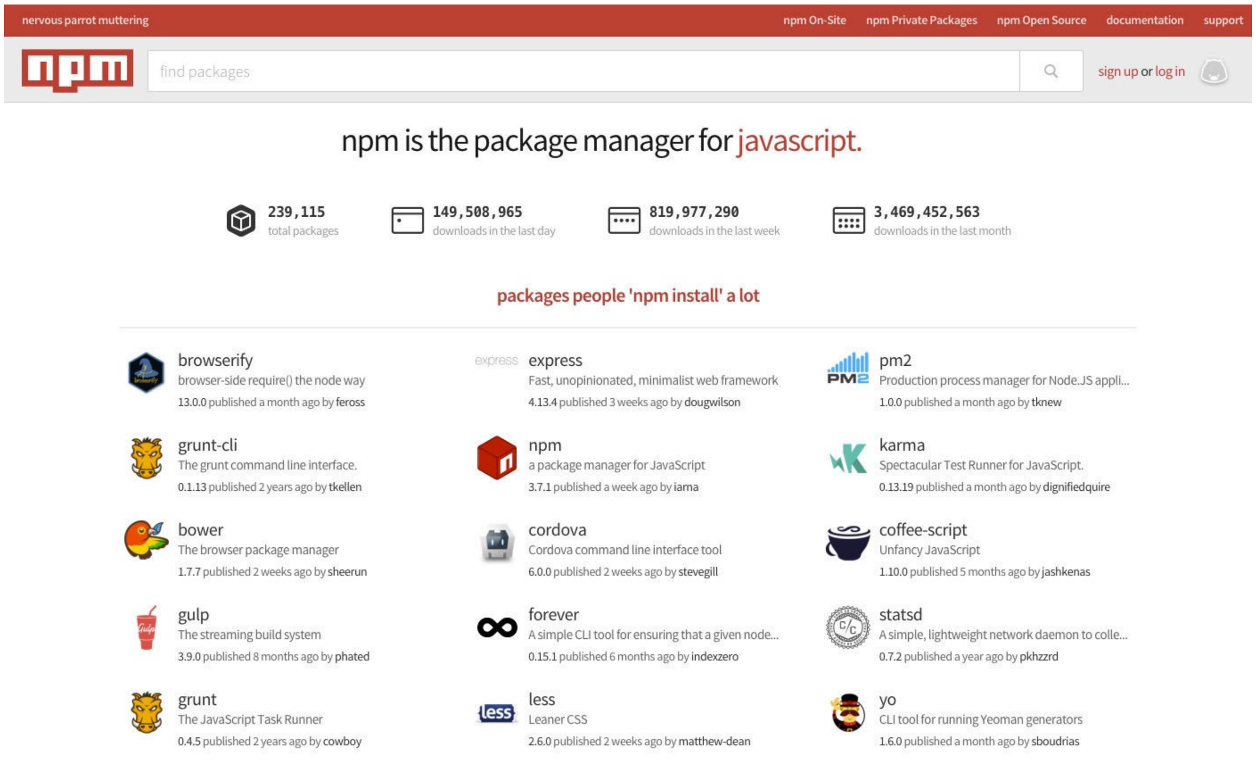
Conclusion

Congrats! Now, besides having the Node.js installed and working, you also learned a new way to manage multiple Node.js versions via NVM. In the next chapter, we are going to explore some important things about NPM (*Node Package Manager*). So, keep reading, because the fun is just starting!

Managing modules with NPM

What does NPM do?

Just like RubyGems from Ruby or Maven from Java, Node.js also has a package manager, which is called NPM *(Node Package Manager)*. It has become so popular throughout the community that since the **0.6.X Node.js version** it was integrated within the main Node.js installer, becoming the standard manager of this platform. This helped the developers at that time, because it made several projects converge to this tool.


nervous parrot muttering

npm On-Site   npm Private Packages   npm Open Source   documentation   support

find packages

sign up or log in

npm is the package manager for javascript.

239,115 total packages
149,508,965 downloads in the last day
819,977,290 downloads in the last week
3,469,452,563 downloads in the last month

packages people 'npm install' a lot

browserify
browser-side require() the node way
13.0.0 published a month ago by feross

express
Fast, unopinionated, minimalist web framework
4.13.4 published 3 weeks ago by dougwilson

pm2
Production process manager for Node.JS appli...
1.0.0 published a month 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 a week ago by iarna

karma
Spectacular Test Runner for JavaScript.
0.13.19 published a month ago by dignifiedquire

bower
The browser package manager
1.7.7 published 2 weeks ago by sheerun

cordova
Cordova command line interface tool
6.0.0 published 2 weeks ago by stevegill

coffee-script
Unfancy JavaScript
1.10.0 published 5 months ago by jashkenas

gulp
The streaming build system
3.9.0 published 8 months ago by phated

forever
A simple CLI tool for ensuring that a given node...
0.15.1 published 6 months ago by indexzero

statsd
A simple, lightweight network daemon to colle...
0.7.2 published a year ago by pkhzzrd

grunt
The JavaScript Task Runner
0.4.5 published 2 years ago by cowboy

less
Leaner CSS
2.6.0 published 2 weeks ago by matthew-dean

yo
CLI tool for running Yeoman generators
1.6.0 published a month ago by sboudrias

npmjs.org

Nowadays, the NPM homepage hosts more than **230k of modules** created by developers, companies and community. **+149 million downloads are made daily and +3.4 million downloads are made monthly**.

Top NPM commands

Use NPM is very easy to use. You can use it to manage your project's dependencies and you create commands for tasks automation purposes, everything is created and managed into the `package.json` file. See bellow a list of the top npm commands:

- `npm init`: displays a simple wizard to help you create and describe your new project;
- `npm install module_name`: install a module;
- `npm install -g module_name`: install a global module;
- `npm install module_name --save`: install a module and add it into the `package.json` file, inside `dependencies`;
- `npm install module_name --save-dev`: install a module and add it into the `package.json` file, inside `devDependencies`;
- `npm list`: lists all the modules installed on the project;
- `npm list -g`: lists all the global modules installed on the OS;
- `npm remove module_name`: uninstall a module from the project;
- `npm remove -g module_name`: uninstall a global module;
- `npm remove module_name --save`: uninstall a module from the project and also remove it from the attribute `dependencies`;
- `npm remove module_name --save-dev`: uninstall a module from the project and also remove it from the attribute `devDependencies`;
- `npm update module_name`: updates the module version;
- `npm update -g module_name`: updates the a global module version;
- `npm -v`: displays the npm current version;
- `npm adduser username`: creates an account to npmjs.org[2];
- `npm whoami`: displays details of your public npm's profile (you must create an account using the previous command);
- `npm publish`: publishes a module into npmjs.org (it's necessary to have an active account first);

Understanding package.json file

All Node.js projects are called modules. But, what is a module? Throughout the book, note that I am going to use the terms module, library and framework, in practice, they all mean the same thing.

The term **module** was taken from the JavaScript's concept, which is a language that works with a **modular architecture**. When we create a Node.js project, that is, a module, this project is followed by a descriptor file of modules, called `package.json`.

[2]https://npmjs.org

This file is essential to a Node.js project. A `package.json` badly written can cause bugs or even prevent your project to be executed, because this file has some key attributes, that are read by both Node.js and the NPM.

To demonstrate it, see bellow a simple example of a `package.json` that describes the main attributes of a module:

```
{
  "name": "my-first-node-app",
  "description": "My first node app",
  "author": "User <user@email.com>",
  "version": "1.2.3",
  "private": true,
  "dependencies": {
    "module-1": "1.0.0",
    "module-2": "~1.0.0",
    "module-3": ">=1.0.0"
  },
  "devDependencies": {
    "module-4": "*"
  }
}
```

With those few attributes, you are able to describe about your module. The attribute `name` is the main one. It defines a programmatic name of the project, this module name will be called via `require("my-first-node-app")` function. It must be written in a clean and short way, offering an abstract about what it will be the module.

The `author` attribute is the one which informs the name and email of the author. You can use the format `Name <email>` in order to websites, such as npmjs.org[3] can read this information correctly.

Another main attribute is the `version`, it defines the current version of the module. It is highly recommended that you use this attribute to allow a module installation via `npm install my-first-node-app` command. The attribute `private` is optional, it is only a boolean that determines if the project is an open-source or is a private project.

Node.js modules work with **three levels of versioning**. For example, the version `1.2.3` is divided into the levels:

1. **Major: X.0.0**
2. **Minor: 0.X.0**
3. **Patch: 0.0.X**

[3]https://npmjs.org

Note the **X** means the current level version to update.

In the previous `package.json` have `4` modules, each one uses a different semantic version. The first one, the `"module-1"`, has a fixed version `1.0.0`. Use this kind of version to install dependencies whose updates can break the project if you change the version.

The `"module-2"` already has a certain update flexibility. It uses the character `"~"` which allows you to update a module as a `patch` level `1.0.x` (it only update the `x` version). Generally, these updates are safe, they brings improvements and bug fixes.

The `"module-3"` updates versions which are greater or equal to `1.0.0` in all the version levels. In many cases, using `">="` can be risky, because the dependency can be updated to a `major` or `minor` level and, consequently, can bring some modifications that can break your application.

The last one, the `"module-4"` uses the `"*"` character. This one will always catch the latest version of the module in any version level. It also can cause troubles in the updates and it has the same behavior as the `modulo-3` versioning. Generally, it is used only in the `devDependencies` which are dependencies used for tests purposes and do not affect the application behavior. **Do not use this kind of version in production environment!**

In case you want to have a more precise control over the versions of your dependencies after their installation in your project, run the `npm shrinkwrap` command. It will lock all dependencies version within the file `npm-shrinkwrap.json` allowing you to have control over the dependencies versions of your project. This command is very used in production's environment, where the versions need to be stricter.

NPM Task automation

You can also automate tasks using the `npm` command. In practice, you can only create new executable commands running `npm run command_name`. To declare those new commands, just create them inside the attribute `scripts` on `package.json` like this example bellow:

```
{
  "name": "my-first-node-app",
  "description": "My first node app",
  "author": "User <user@email.com>",
  "version": "1.2.3",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "clean": "rm -rf node_modules",
    "test": "node test.js"
  },
  "dependencies": {
    "module-1": "1.0.0",
    "module-2": "~1.0.0",
    "module-3": ">=1.0.0"
  },
  "devDependencies": {
    "module-4": "*"
  }
}
```

Note that this new `package.json` created three scripts: `start`, `clean` and `test`. These scripts are executable now via their commands: `npm run start`, `npm run clean` and `npm run test`. As shortcut, only the `start` and `test` commands can be executed by the alias: `npm start` and `npm test`. Within the scripts, you can run both the commands `node`, `npm` or any other global command from the OS. The `npm run clean` is an example of a global command from the `bash`, which internally runs the command: `rm -rf node_modules` to deletes all files from the `node_modules` folder.

Conclusion

If you reached this point, you have learned the basics about managing dependencies and automating tasks using npm. It is highly important to understand these concepts, because it will be frequently used throughout this book.

Fasten your seatbelts and get ready! In the next chapter we are going to start our API RESTful pilot project, which is going to be worked on throughout the next chapters. As spoiler, I am going to tell you about what this project will be.

In the next chapters, we are going to create a simple API REST system to manage tasks. Not only an API is going to be developed but also we'll create a web application to consume data from this API. All development is going to happen using some Node.js popular frameworks: Express to create the API resources, Sequelize to persist data in a relational database, Passport to deal with user authentications and a lot more.

Building an API

Now that we have the Node.js environment installed and ready to work, we are going to explore the creation of our REST API application. This kind of application is currently being developed by many projects and companies, because it brings as an advantage the creation of an application focused only on feeding any client-side application with data. Nowadays, it's quite common to create web and or mobile apps which consumes data from an API or more than one APIs. This makes that several kinds of client applications consult the same server focused on deal with data. Besides, it also allows that each application – client or server – to be worked by different teams.

First, we are going to build an API, however, throughout to the last chapters, we will build a simple web client-side application to consume data from the our API. To start developing the API, we are going to use a very popular web framework, called Express.

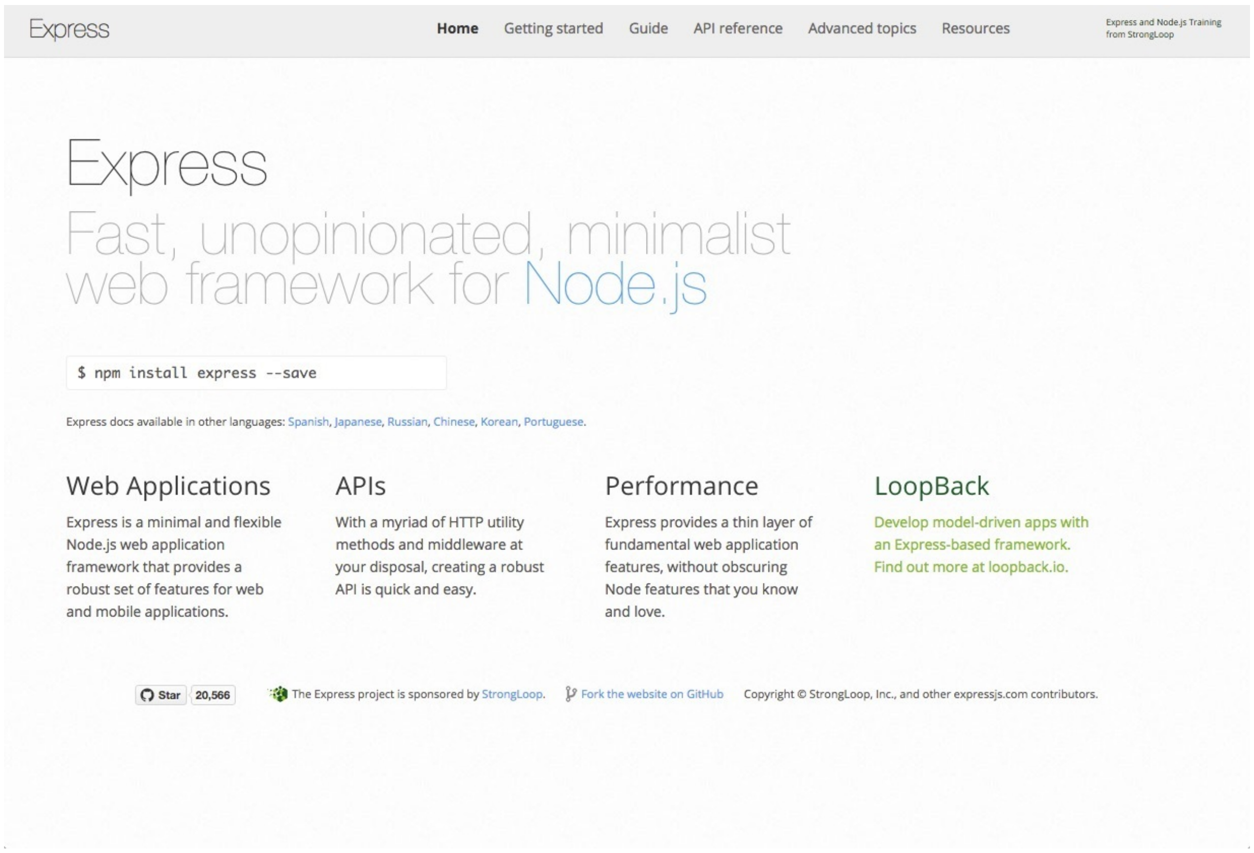
Introduction to Express

Express is a minimalist web framework that was highly inspired in Sinatra framework from Ruby language. With this module, you can create from small applications to large and complex ones. This framework allows you to build APIs and also to create simple web sites.

It's focused to work with `views`, `routes` and `controllers`, only `models` is not handled by this framework, giving you the freedom to use any persistence framework without creating any incompatibility or conflict in the application. This is a big advantage, because there're a lot of ODM *(Object Data Mapper)* and also ORM *(Object Relational Mapping)* available. You can use anyone with Express without problem, you just need to load this kind of module, write some models and use them inside the `controllers`, `routes` or `views`.

Express allows the developers freely arrange the project's code, that is, there're no inflexible conventions in this framework, each convention can be created and applied by yourself. This makes Express flexible to be used on both small and large applications, because not always it's necessary to apply a lot of conventions into a small project.

You can also reapply conventions from other frameworks. Ruby On Rails is an example of a framework full of conventions that are worth reapplying some of their technical features. This independence forces the developer to fully understand how each application structure works.



Express

Fast, unopinionated, minimalist web framework for Node.js

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

Express docs available in other languages: Spanish, Japanese, Russian, Chinese, Korean, Portuguese.

Web Applications

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

APIs

With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.

Performance

Express provides a thin layer of fundamental web application features, without obscuring Node features that you know and love.

LoopBack

Develop model-driven apps with an Express-based framework. Find out more at loopback.io.

The Express project is sponsored by StrongLoop.

Copyright © StrongLoop, Inc., and other expressjs.com contributors.

expressjs.com

To sum up, you can see a list of Express main features bellow:

- Robust routing;
- Easily integratable with a lot of template engines;
- Minimalist code;
- Works with middlewares concept;
- A huge list of 3rd-party middlewares to integrate;
- Content negotiation;
- Adopt standards and best practices of REST APIs;

Getting started the pilot project

What about creating a project in practice? From this chapter on, we are going to explore some concepts on how to create a REST API using some Node.js frameworks.

Our application is going to be a simple task manager, this will be divided into two projects: API and Web app.

Our API will be called **NTask** (Node Task) and it'll have the following features:

- List of tasks;
- Create, delete and update a task;
- Create, delete and update a user data;
- User authentication;
- API documentation page;

Pilot project source code

If you are curious to check out the source code of all projects that are going to be explored in this book, just access this link:

github.com/caio-ribeiro-pereira/building-apis-with-nodejs[4]

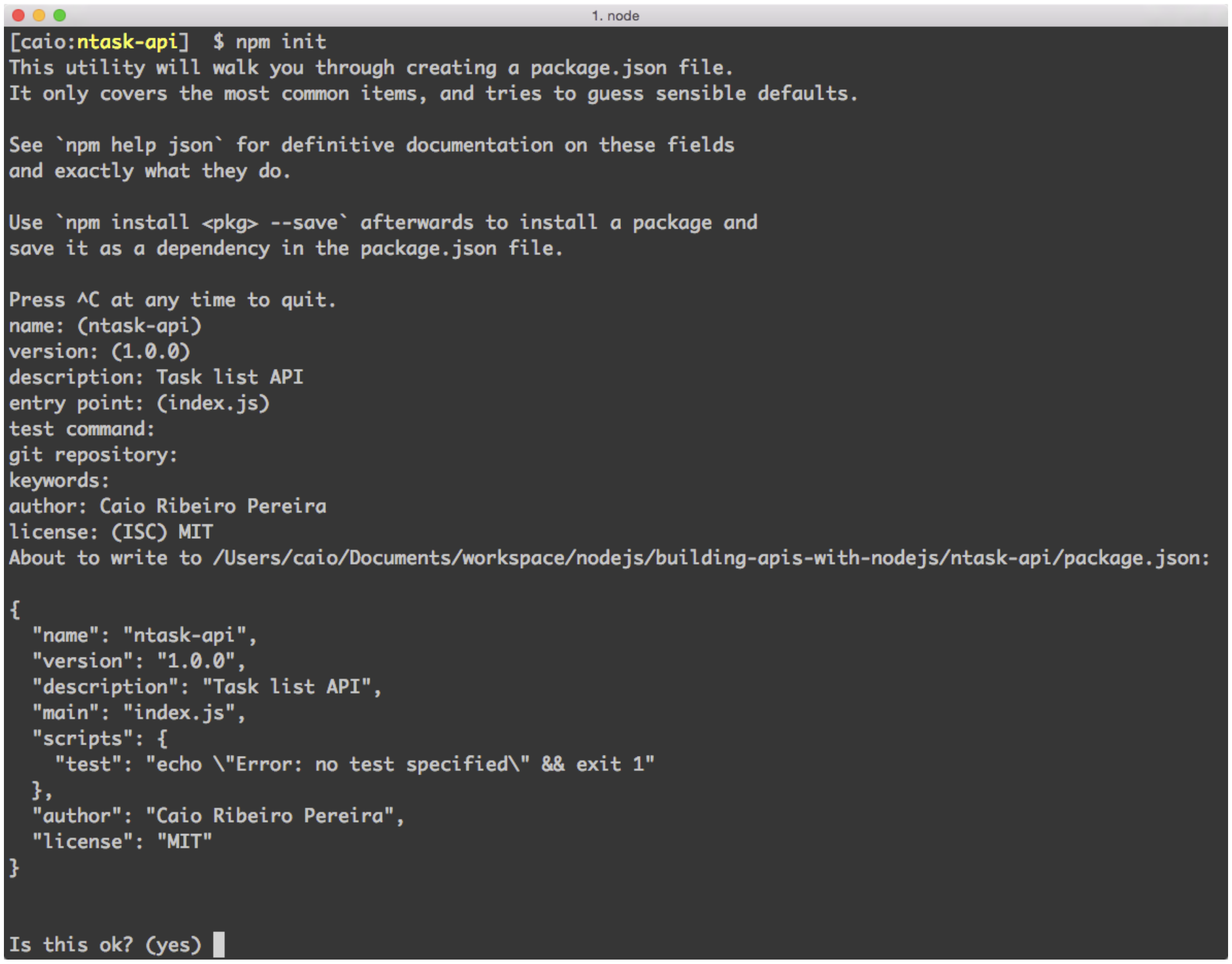
In this application, we'll explore the main resources to create an REST API on Node.js and, throughout the chapters; we are going to include new important frameworks to help the developing.

To begin, let's create our first project named `ntask-api`, running these commands:

```
mkdir ntask-api
cd ntask-api
npm init
```

You need to answer the `npm init` command wizard as you like or you can copy this result below:

[4]https://github.com/caio-ribeiro-pereira/building-apis-with-nodejs



```
[caio:ntask-api]  $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (ntask-api)
version: (1.0.0)
description: Task list API
entry point: (index.js)
test command:
git repository:
keywords:
author: Caio Ribeiro Pereira
license: (ISC) MIT
About to write to /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-api/package.json:

{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "Task list API",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Caio Ribeiro Pereira",
  "license": "MIT"
}


Is this ok? (yes)
```

Describing the project using npm init

In the end, the `package.json` will be created with these attributes:

```json
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "Task list API",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Caio Ribeiro Pereira",
  "license": "ISC"
}
```

The currenty Node.js version does not support ES6 perfectly, but we can use a module that emulates some of the resources from ES6/ES7 to make our codes cooler. To do this, we are going to install the `babel`. It’s JavaScript transpiler responsible to convert a ES6/ES7 codes to a ES5 code only when the JavaScript runtime doesn’t recognize some ES6/ES7 features.

To use all features from ES6/ES7, we are going to install `babel-cli` and `babel-preset-es2015` modules in the project, running this command:

```
npm install babel-cli@6.5.1 babel-preset-es2015@6.5.0 --save
```

Now you need to link the preset `babel-preset-es2015` to be recognized by the `babel-cli`, to do this, just create the file `.babelrc` with this simple code:

```
{
  "presets": ["es2015"]
}
```

After that, we’ll to turbocharge the `package.json`. First we are going to remove the fields `license` and `scripts.test`. Then, include the field `scripts.start` to enable the alias command `npm start`. This command will be responsible to start our API server via Babel transpiler using the command `babel-node index.js`. See bellow how our `package.json` is going to look like:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "Task list API",
  "main": "index.js",
  "scripts": {
    "start": "babel-node index.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0"
  }
}
```

Now we have a basic configuration and description about our project, and each information is inside the `package.json`. To start off, let’s install the `express` framework:

```
npm install express@4.13.4 --save
```

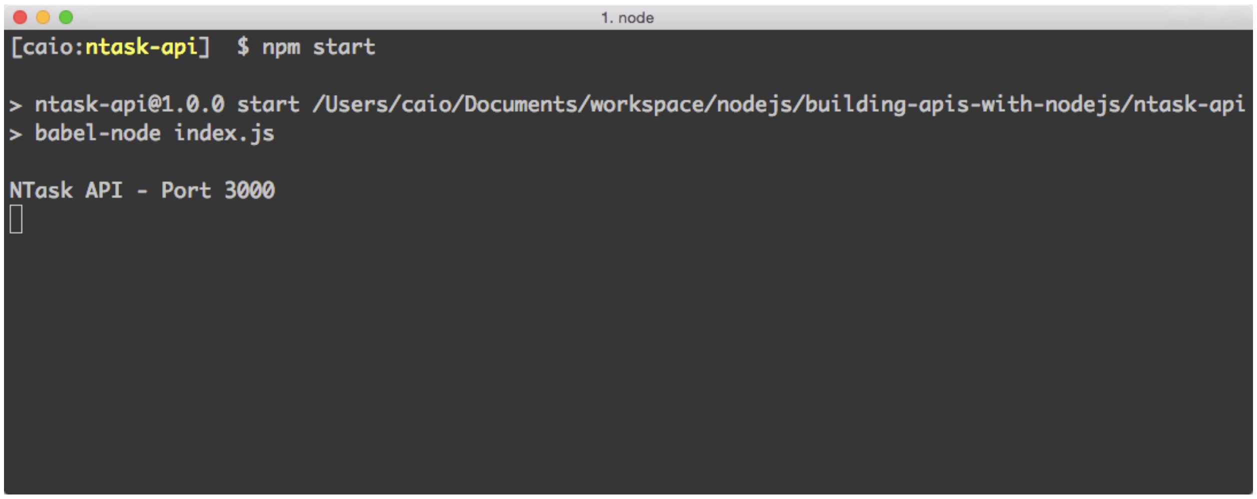
Installing Express, we’ll create our first code. This code is going to load the `express` module, create a simple endpoint using the `GET /` via function `app.get("/")` and start the server in the port `3000` using the function `app.listen()`. To do this, create the `index.js` file using this code:

```
import express from "express";

const PORT = 3000;
const app = express();

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

To test this code and especially check if the API is running, start the server running:

```
npm start
```

Your application must display the following message on the terminal:

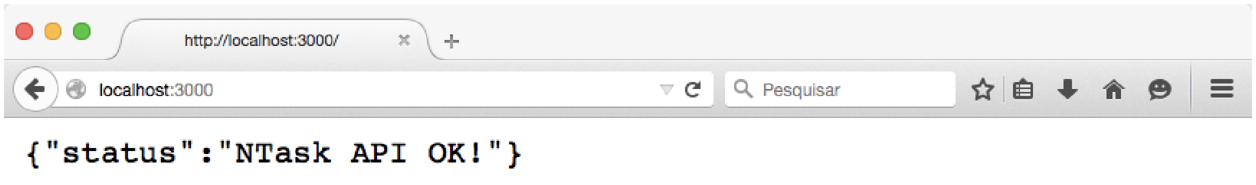


```
[caio:ntask-api]  $ npm start

> ntask-api@1.0.0 start /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-api
> babel-node index.js

NTask API - Port 3000
```

Starting the API

Then, open your browser and go to: localhost:3000[5]

If nothing goes wrong, a JSON content will be displayed, similar to the image bellow:



```
{"status":"NTask API OK!"}
```

JSON status message

[5]http://localhost:3000

Implementing a simple and static resource

REST APIs works with the concept of creates and manipulates **resources**. These resources are entities that are used for queries, entries, updating and deleting data, by the way, everything is based on manipulating data from resources.

For example, our application will have as the main resource the entity named `tasks`, which will be accessed by the endpoint `/tasks`. This entity is going to have some data that describes what kind of information can be persisted in this resource. This concept follows a same line as data modeling, the only difference is that the resources abstract the data source. So, a resource can return data from different sources, such as databases, static data or any data from external systems, it can return from another external API too.

An API aims to address and unify the data to, in the end, build and show a resource. Initially, we are going to work with static data, but throughout the book we will make some refactorings to integrate a database.

For while, static data will be implemented only to mold an endpoint. To mold our API, we are going to include a route via `app.get("/tasks")` function, which is going to return only a static JSON via `res.json()` function, which is responsible to render a json content as output. Here is how these modifications are going to look like in the `index.js` file:

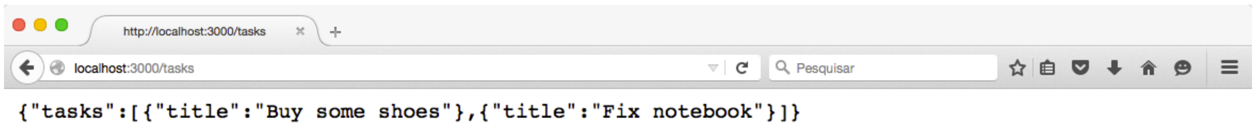
```
import express from "express";

const PORT = 3000;
const app = express();

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.get("/tasks", (req, res) => {
  res.json({
    tasks: [
      {title: "Buy some shoes"},
      {title: "Fix notebook"}
    ]
  });
});

app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

To test this new endpoint, restart the application typing **CTRL+C** or **Control+C** (if you are a MacOSX user) and then, run `npm start` again.

Attention: Always follow this same step above to restart the application correctly.

Now we have a new endpoint available to access via the address: localhost:3000/tasks[6]

Once you open it, you will have the following result:



```
{"tasks":[{"title":"Buy some shoes"},{"title":"Fix notebook"}]}
```

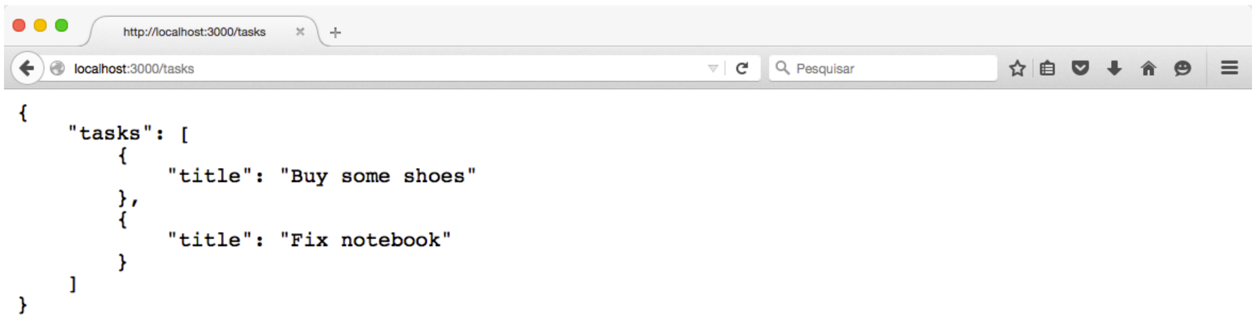
Listing tasks

In case you want that your results to return as a formatted and tabbed JSON output, include in the `index.js` the following configuration: `app.set("json spaces", 4);`. See below where to do it:

```
import express from "express";

const PORT = 3000;
const app = express();

app.set("json spaces", 4);

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.get("/tasks", (req, res) => {
  res.json({
    tasks: [
      {title: "Buy some shoes"},
      {title: "Fix notebook"}
    ]
  });
});

app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

Now, restart the server and see a result more elegant:

[6]http://localhost:3000/tasks



```
{
    "tasks": [
        {
            "title": "Buy some shoes"
        },
        {
            "title": "Fix notebook"
        }
    ]
}
```

Listing tasks with formatted JSON

Arranging the loading of modules

Indeed, write all endpoints into `index.js` won't be a smart move, especially if your application has a lot of them. So, let's arrange the directories and the loading of all codes according to their responsibilities.

We are going to apply the MVR *(Model-View-Router)* pattern to arrange this things. To do it, we'll use the `consign` module, which allows to load and inject dependencies easily. So, let's install!

```
npm install consign@0.1.2 --save
```

With this new module installed, let's migrate the endpoints from `index.js` file creating two new files into the new directory named `routes`. To do this, create the file `routes/index.js`:

```
module.exports = app => {
  app.get("/", (req, res) => {
    res.json({status: "NTask API"});
  });
};
```

Move the endpoint function `app.get("/tasks")` from `index.js` to his new file `routes/tasks.js`:

```
module.exports = app => {
  app.get("/tasks", (req, res) => {
    res.json({
      tasks: [
        {title: "Buy some shoes"},
        {title: "Fix notebook"}
      ]
    });
  });
};
```

To finish this step, edit the `index.js` to be able to load those routes via `consign` module and start the server:

```
import express from "express";
import consign from "consign";

const PORT = 3000;
const app = express();

app.set("json spaces", 4);

consign()
  .include("routes")
  .into(app);

app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

That's it, we have just arranged the loading of all routes. Note that, at this point we are only focusing on working with **VR** (view and router) from **MVR** pattern. In our case, the JSON outputs are considered as **views** which are provided by the **routes**. The next step will be arranging the **models**.

Let's create the `models` directory and, back to `index.js` add one more `include()` function inside the `consign()` to allow the loading of the `models` before the `routes`. To make this modification clearer, edit the `index.js`:

```
import express from "express";
import consign from "consign";

const PORT = 3000;
const app = express();

app.set("json spaces", 4);

consign()
  .include("models")
  .then("routes")
  .into(app);

app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

In this moment, the `consign()` function won't load any model, because the directory `models` doesn't exists. To fill this gap, let's temporarily create a model with static data, just to finish this step. To do this, create the file `models/tasks.js` and filling with these codes:

```
module.exports = app => {
  return {
    findAll: (params, callback) => {
      return callback([
        {title: "Buy some shoes"},
        {title: "Fix notebook"}
      ]);
    }
  };
};
```

Initially, this model is going to only have the function `Tasks.findAll()`, which is going to receive two arguments: `params` and `callback`. The variable `params` will not be used in this moment, but it serves as a base to send some SQL query filters, something that we are going to talk about in details in the next chapters. The second argument is the callback function, which returns asynchronously a static array of tasks.

To call it in the `routes/tasks.js` file, you are going to load this module via `app` variable. After all, the modules added by `consign()` function are injected into a main variable, in our case, injected into `app` via `consign().into(app);`. To see how to use a loaded module, in this case, we'll use the `app.models.tasks` module, edit the `routes/tasks.js` following this code:

```
module.exports = app => {
  const Tasks = app.models.tasks;
  app.get("/tasks", (req, res) => {
    Tasks.findAll({}, (tasks) => {
      res.json({tasks: tasks});
    });
  });
};
```

Note that the function `Tasks.findAll()` has in the first parameter an empty object. This is the value of the `params` variable, which in this case, there is no need to include parameters to filter the tasks list.

The callback of this function returns the `tasks` variable, which was created to return, for while, some static values from his model. So, at this point, we are sure that `tasks` is going to return an static array with some tasks descriptions.

To finish these refactorings, let's create a file, which is going to load all the middlewares and specific settings of Express. Currently, we only have a simple configuration about JSON's format, which occurs via function `app.set("json spaces", 4)`. But, we are going to include another settings, in this case, it will be the server port calling the `app.set("port", 3000)` function.

Throughout this book, we are going to explore a lot of new middlewares and settings to customize our API server. So, I already recommend to prepare the house for new visitors.

Create the file `libs/middlewares.js` and write theses codes below:

```
module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
};
```

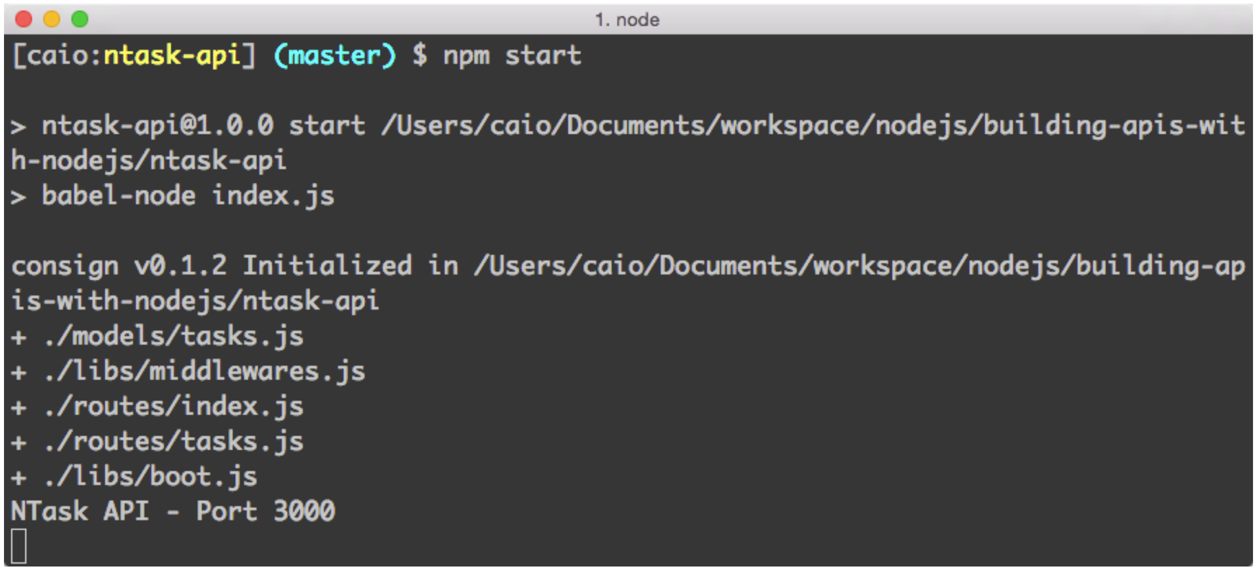
To simplify the server boot, create the file `libs/boot.js` that is going to be responsible for the server initialization via `app.listen()` function, this time, we'll use `app.get("port")` instead the old constant `PORT`, see the code below:

```
module.exports = app => {
  app.listen(app.get("port"), () => {
    console.log(`NTask API - Port ${app.get("port")}`);
  });
};
```

To finish this chapter, let's load the `libs/boot.js` inside the module's structure provided by `consign` and remove the old `const PORT`, `app.set()` and `app.listen()` declarations to simplify this main file. To do this, just edit the `index.js`:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("models")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

To test everything, restart the server and access the endpoint again: localhost:3000/tasks[7]



```
[caio:ntask-api] (master) $ npm start

> ntask-api@1.0.0 start /Users/caio/Documents/workspace/nodejs/building-apis-wit
h-nodejs/ntask-api
> babel-node index.js

consign v0.1.2 Initialized in /Users/caio/Documents/workspace/nodejs/building-ap
is-with-nodejs/ntask-api
+ ./models/tasks.js
+ ./libs/middlewares.js
+ ./routes/index.js
+ ./routes/tasks.js
+ ./libs/boot.js
NTask API - Port 3000
```

Listing loaded modules

To make sure everything is ok, no error should occur and all the tasks data must be displayed normally.

Conclusion

Mission complete! In the next chapter we'll expand our project implementing new functions to manage tasks dynamically using a database and Sequelize.js framework.

[7]http://localhost:3000/tasks

Working with SQL databases

Introduction to SQLite3 and Sequelize

In the last chapter, we created simples routes for the application to list tasks via static data model. That was enough to start and explore some basic concepts about API resources.

Now, we are going to work deeper on using a relational database, which is going to be used to create a dynamic management of tasks and users. To simplify our examples, we are going to use **SQLite3**. It is preinstalled database on Linux, Unix and MacOSX, so there's no need to set it up. However, in case you are using Windows, you can easily install it following the instructions on the link bellow:

https://www.sqlite.org/download.htm

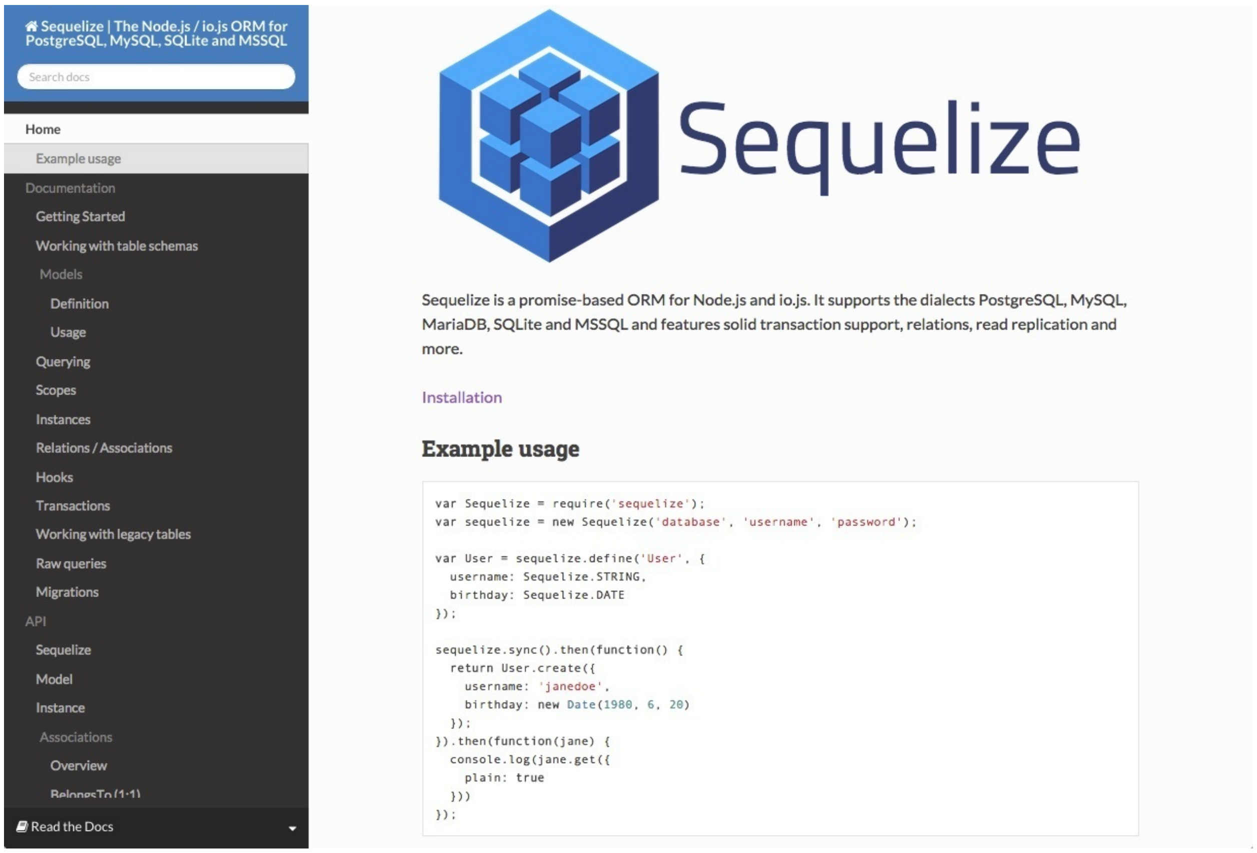
SQLite3 logo

SQLite3 is a database that keeps all the data in a `.sqlite` extension's file. It has an interface based on SQL language very similar to other databases and is present not only on desktop systems but on mobile applications as well.

On Node.js, there are several frameworks that works with **SQLite3**. On our application, we'll use the **Sequelize.js** module, which is a complete module and has a nice interface and is very easy to work. On it, it is possible to manipulate data using SQL commands (or not) and it also supports the main SQL databases, such as: **Postgres**, **MariaDB**, **MySQL**, **SQL Server and SQLite3**.

Sequelize.js logo

Sequelize is an ORM *(Object Relational Mapper)* for Node.js. It’s **Promises-based** framework. Currently, it is on **3.19.0 version** and has a lot of features like transactions, modeling tables, tables relationship, database replication for reading mode and more.



Sequelize is a promise-based ORM for Node.js and io.js. It supports the dialects PostgreSQL, MySQL, MariaDB, SQLite and MSSQL and features solid transaction support, relations, read replication and more.

Installation

Example usage

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('database', 'username', 'password');

var User = sequelize.define('User', {
  username: Sequelize.STRING,
  birthday: Sequelize.DATE
});

sequelize.sync().then(function() {
  return User.create({
    username: 'janedoe',
    birthday: new Date(1980, 6, 20)
  });
}).then(function(jane) {
  console.log(jane.get({
    plain: true
  }))
});
```

Sequelize.js homepage

The read the full documentation take a look this link:

http://sequelizejs.com

Setting up Sequelize

To install Sequelize and SQLite3 wrapper, execute on the terminal this command:

```
npm install sequelize@3.19.2 sqlite3@3.1.1 --save
```

With these two modules installed, we already have the dependencies that we need for the database connection with the API. Now, let's create a connection settings file between Sequelize and SQLite3. To do it, create the file `libs/config.js` using these parameters:

- `database`: defines the database name;
- `username`: informs the username of access;
- `password`: informs the username's password;
- `params.dialect`: informs which database will be used (`sqlite`, `mysql`, `postgres`, `mariadb`, `mssql`);
- `params.storage`: is a specific attribute for only SQLite3, it defines the directory where the database files will be recorded;
- `params.define.underscored`: standardize the tables fields' name in lowercase letters with underscore;

See bellow how the file is going to look like:

```
module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    define: {
      underscored: true
    }
  }
};
```

After create this settings file, we're going create a code, which will be responsible to connects with the database. This connection code will adopt the **Singleton pattern** to ensure that Sequelize connection will be instantiated only once. This is going to allow us to load this module innumerous time via a single database's connection.

To do this, create the `db.js` code following the step below:

```
import Sequelize from "sequelize";
const config = require("./libs/config.js");
let sequelize = null;

module.exports = () => {
  if (!sequelize) {
    sequelize = new Sequelize(
      config.database,
      config.username,
      config.password,
      config.params
    );
  }
  return sequelize;
};
```

Done! To start this connection module, let's include it inside `consign()` function. Now the `db.js` will be the first module to be loaded. To do this, edit `index.js`:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("db.js")
  .then("models")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

To finish the Sequelize.js setup, let's implement a sync simple function between Sequelize and database. This sync function performs, if necessary, alterations on database tables, according to what is going to be setup on the models. Let's include the `app.db.sync()` function, to ensure this action will be executed before the server starts. This refactoring needs to be written into `libs/boot.js` file:

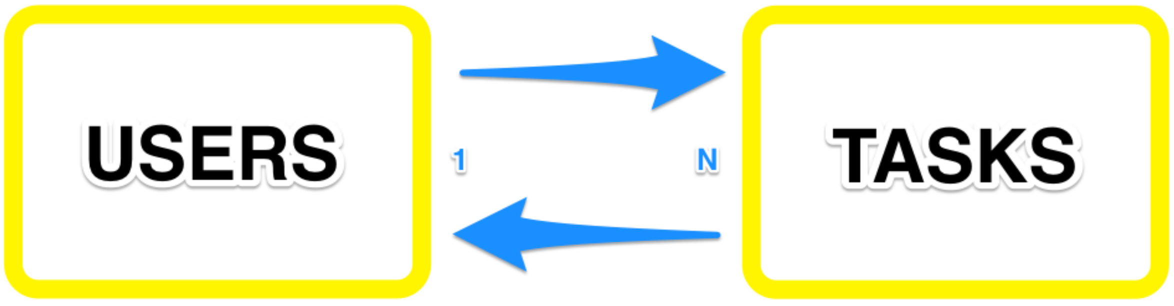
```
module.exports = app => {
  app.db.sync().done(() => {
    app.listen(app.get("port"), () => {
      console.log(`NTask API - Port ${app.get("port")}`);
    });
  });
};
```

To test those modifications, restart the server. If everything is ok, your application must work as it was working before, after all, no visible modification was made, only some adjustments were made to establish a connection between the API and SQLite3. In the next section, we'll create all necessary models for our application, and this is gonna be a huge impact for our project.

Creating models

Up to this point, the only model in the API is the `model/tasks.js`, which is static model and is not connected with any database. In this section, we are going to explore the main Sequelize features to create models that performs as tables in the database and as resources in the API.

Our application will have two simple models: `Users` and `Tasks`. The relationship between them will be **Users 1-N Tasks**, similar to the image below:



NTask models relationship

To work with this kind of relationship, will be used the Sequelize's functions: `Users.hasMany(Tasks)` (on `models/users.js`) and `Tasks.belongsTo(Users)` (on `models/tasks.js`). These associations will be encapsulated within a model's attribute called `classMethods`, which allows us to include static functions to the model. In our case, we are going to create the `associate` function within the `classMethods` of each model.

Model: Tasks

To start off, let's modify the `models/tasks.js` file, applying the following code:

```
module.exports = (sequelize, DataType) => {
  const Tasks = sequelize.define("Tasks", {
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    title: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    done: {
      type: DataType.BOOLEAN,
      allowNull: false,
      defaultValue: false
    }
  }, {
    classMethods: {
      associate: (models) => {
        Tasks.belongsTo(models.Users);
      }
    }
  });
  return Tasks;
};
```

The function `sequelize.define("Tasks")` is responsible to create or change a table. This happens only when Sequelize syncs with the application during the boot time (in our case, this happens on `libs/boot.js` file via `app.db.sync()` function). The second parameter is an object, their attributes respectively represent the fields of a table, and their values are sub attributes whose describes the data type their fields

On this model, the field `id` is an integer via `DataType.INTEGER. It represents a primary key (`primaryKey: true`) and its value is auto incremented (`autoIncrement: true‘) on each new record.

The field `title` is a string (`DataType.STRING`). This field has the attribute `allowNull: false` to block null values and is a validator field as well, which verifies if the string is not empty (`validate.notEmpty: true`). The field `done` is a boolean (`DataType.BOOLEAN`) which doesn't allow null values (`allowNull: false`). By the way, if a value is not informed in this field, it will be registered as `false` via `defaultValue: false` attribute.

At last, we have a third parameter, which allows including static functions within attribute `classMethods`. The `associate(models)` function was created to allow models' relationships. In this case, the relationship was established via `Tasks.belongsTo(models.Users)` function, cause this is a **Tasks 1-N Users** relationship.

WARNING: The model `Users` wasn't created yet. So, if you restart the server right now an error will happen. Stay calm, keep reading and coding until the end of this chapter to see the result in the end.

Model: Users

To complete our database modeling, let's create a model that will represent the users of our application. You need to create the `models/users.js` file with the following model definitions:

```
module.exports = (sequelize, DataType) => {
  const Users = sequelize.define("Users", {
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    name: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    password: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    email: {
      type: DataType.STRING,
      unique: true,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    }
  }, {
    classMethods: {
      associate: (models) => {
        Users.hasMany(models.Tasks);
      }
    }
  });
  return Users;
};
```

This time, the modeling of the table `Users` was very similar to `Tasks`. The only difference was the inclusion of the `unique: true` attribute, inside the field `email`, to ensure that repeated emails won't be allowed.

After finishing this step, we are going to change some codes in the project, so it can load these models correctly and run their respective relationship functions in the database. To start off, let's modify some of the modules loading on `index.js`. First, we are going to order the loading of the modules to the `libs/config.js` be loaded first, then the `db.js` and also remove the loading of the `models` directory from `consign`, because now all models are loaded from `db.js`. See how the code must look like:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("libs/config.js")
  .then("db.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

The reason why the directory `models` was deleted from `consign()` is that now, all models will be loaded directly by `db.js` file, via `sequelize.import()` function. After all, if you go back to the models codes you will notice that two new attributes within `module.exports = (sequelize, DataType)` appeared. Those are going to be magically injected via `sequelize.import()`, which is responsible for loading and defining the models. So, let's refactory the `db.js` file, to import and load all models correctly:

```
import fs from "fs";
import path from "path";
import Sequelize from "sequelize";

let db = null;

module.exports = app => {
  if (!db) {
    const config = app.libs.config;
    const sequelize = new Sequelize(
      config.database,
      config.username,
      config.password,
      config.params
    );
    db = {
      sequelize,
      Sequelize,
      models: {}
    };
    const dir = path.join(__dirname, "models");
    fs.readdirSync(dir).forEach(file => {
      const modelDir = path.join(dir, file);
      const model = sequelize.import(modelDir);
      db.models[model.name] = model;
    });
    Object.keys(db.models).forEach(key => {
      db.models[key].associate(db.models);
    });
  }
  return db;
};
```

This time, the code got a little complex, didn't it? But, its features are pretty cool! Now we can use the database settings via `app.libs.config` object. Another detail is in the execution of the nested function `fs.readdirSync(dir).forEach(file)` which basically returns a array of strings referring to the files names from the directory `models`. Then, this array is iterated to import and load all models using the `sequelize.import(modelDir)` function and, then, inserted in this module inside the structure `db.models` via `db.models[model.name] = model`.

After loading all models, a new iteration happens through `Object.keys(db.models).forEach(key)` function, which basically executes the function `db.models[key].associate(db.models)` to ensure the models' relationship.

To finish the adjustments, we need to write a simple modification in the `libs/boot.js` file, changing from the `app.db.sync()` function to `app.db.sequelize.sync()`:

```
module.exports = app => {
  app.db.sequelize.sync().done(() => {
    app.listen(app.get("port"), () => {
      console.log(`NTask API - Port ${app.get("port")}`);
    });
  });
};
```

Then, let's modify the `routes/tasks.js` to load the `Tasks` model correctly using `app.db.models.Tasks` and modify the function `Tasks.findAll()` to his promises-based function provided by Sequelize. See below how it must look like:

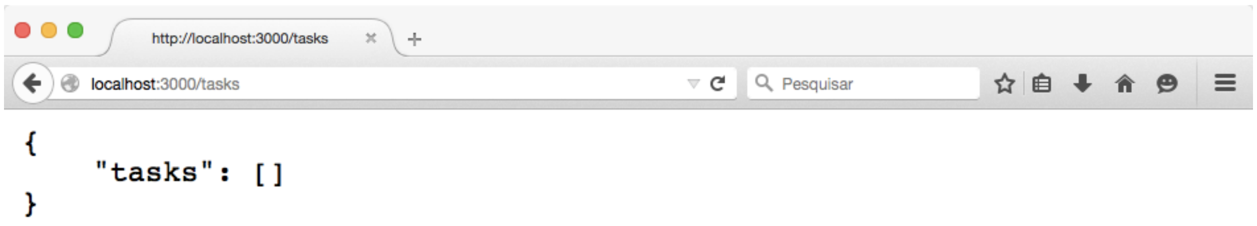
```
module.exports = app => {
  const Tasks = app.db.models.Tasks;
  app.get("/tasks", (req, res) => {
    Tasks.findAll({}).then(tasks => {
      res.json({tasks: tasks});
    });
  });
};
```

Conclusion

Finally we have finished the Sequelize.js adaptation into our API. To test it, just restart the server and go to the address:

http://localhost:3000/tasks

This time it is going to display a JSON object with empty tasks.



```
{
    "tasks": []
}
```

Now, the tasks list is empty!

But don’t worry, in the next chapter we’ll implement the main endpoints to perform a complete CRUD in the API. So just keep reading!

CRUDify API resources

In this chapter, we'll explore more the usage of the new functions from Sequelize and also organize the API's routes and some middlewares of Express. The CRUD *(Create, Read, Update, Delete)* will be built using the models: `Tasks` and `Users`.

Organizing tasks routes

To start this refactoring, let's explore the main HTTP methods to CRUDify our application. In this case, we are going to use the functions `app.route("/tasks")` and `app.route("/tasks/:id")` to define the path: `"/tasks"` and `"/tasks/(task_id)"`.

These functions allow to use nested functions, reusing the routes through all HTTP methods provided by these Express functions below:

- `app.all()`: is a middleware that is executed via any HTTP method call;
- `app.get()`: executes the method `GET` from HTTP, it is used to search and get some set of data from one or multiple resources;
- `app.post()`: executes the method `POST` from HTTP, it is semantically used to send and persist new data into a resource;
- `app.put()`: executes the method `PUT` from HTTP, very used to update data of a resource;
- `app.patch()`: executes the method `PATCH` do HTTP, it has similar semantics to `PUT` method, but is recommended only to update some attributes of a resource, and not the entire data of a resource;
- `app.delete()`: executes the method `DELETE` from HTTP, as his name implies to delete a particular data of a resource.

To better understand the use of these routes, we are going to edit the `routes/tasks.js`, applying the needed functions to CRUDify the tasks resource.

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .all((req, res) => {
      // Middleware for pre-execution of routes
    })
    .get((req, res) => {
      // "/tasks": List tasks
    })
    .post((req, res) => {
      // "/tasks": Save new task
    });

  app.route("/tasks/:id")
    .all((req, res) => {
      // Middleware for pre-execution of routes
    })
    .get((req, res) => {
      // "/tasks/1": Find a task
    })
    .put((req, res) => {
      // "/tasks/1": Update a task
    })
    .delete((req, res) => {
      // "/tasks/1": Delete a task
    });
};
```

With this basic structure, we already have an outline of the needed routes to manage tasks. Now, we can implement their logics to correctly deal with each action of our tasks CRUD.

On both endpoints (`"/tasks"` e `"/tasks/:id"`) we are going to deal with some rules inside their middlewares via `app.all()` function to avoid some invalid access when if somebody sends the task `id` inside request's body. This is going to be a simple modification, see below how must look like:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  // Continuation of routes...

app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  // Continuation of routes...
```

We are practically ensuring the exclusion of the `id` attribute within the request's body, via `delete req.body.id`. This happens because, on each request functions, we are going to use `req.body` as a parameter for Sequelize.js functions, and the attribute `req.body.id` could overwrite the `id` of a task, for example, on `update` or `create` a task.

To finish the middleware, warning that it should perform a corresponding function to an HTTP method, you just need to include in the end of callback the `next()` function to warn the Express router that now it can execute the next function on the route or the next middleware below.

Listing tasks via GET

We already have a basic middleware for treating all the task's resources, now let's implement some CRUD functions, step-by-step. To start off, we'll the `app.get()` function, which is going to list all data from `tasks` models, running the `Tasks.findAll()` function:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findAll({})
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

In this first implementation, we are listing all the tasks running: `Tasks.findAll({})`. Although it is a bad practice to list all data, we are only using it for didactic matters. But chill out that throughout the chapters some arguments will be implemented to make this list of tasks more specific. The search results occurs via `then()` function and if any problem happens, you can handle them via `catch()` function. An important detail about API's development is to treat all or the most important results using the correct HTTP status code, in this case a successful result will return by default the `200 - OK` status. On our API, to simplify things a lot of common errors will use the `412 - Precondition Failed` status code.

About HTTP status

There is no rules to follow in the HTTP status definition; however, it's advisable to understand the meaning of the each status to make your server's response more semantic. To learn more about some HTTP status code, take a look this website:

www.w3.org/Protocols/rfc2616/rfc2616-sec10.html[8]

Creating tasks via POST

There is no secret in the implementation of POST method. Just use the `Tasks.create(req.body)` function and then treat their promises callbacks: `then()` and `catch()`.

One of the most interesting things on Sequelize is the sanitization of parameters you send to a model. This is very good, because if the `req.body` have some attributes not defined in the model, they will be removed before inserts or updates a model. The only problem is the `req.body.id` that could modify the `id` of a model. However, this was already handled by us when we wrote a simple rules into `app.all()` middleware. The implementation of this route will be like this piece of code:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findAll({})
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .post((req, res) => {
```

[8]http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

```
    Tasks.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
```

Finding a task via GET

Now we are gonna to treat the `/tasks/:id` endpoints. To do this, let's start by the implementation of the `app.route("/tasks/:id")` function, which have the same middleware logic as the last `app.all()` function.

To finish, we'll use the function `Tasks.findOne({where: req.params})`, which executes, for example, `Tasks.findOne({where: {id: "1"}})`. It does an single search of tasks based on the task `id`, in case there isn't a task, the API will respond the `404 - Not Found` status code via `res.sendStatus(404)` function. See how it must look like:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

Updating a task via PUT

Now we are going to implement a function to update a task in the database. To do this, there is no secret, just use the `Task.update()` function which the first parameter you have included an updated object and, in the second one, an object with parameters to find the current task to be updated. These functions will return a simple array with a number of changes made. But this information won't be very useful as response, so to simplify things, let's force the `204 - No Content` status code using the `res.sendStatus(204)` function, which means that the request was successful done and no content will be returned in the response. See below how this implementation looks like:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .put((req, res) => {
    Tasks.update(req.body, {where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

Just like the `Tasks.create()` function, the `Tasks.update` cleans the fields that are not on its own model, so there is no problem to send `req.body` directly as parameter.

Deleting a task via DELETE

To finish, we have to implement a route to delete a task, and once more, there is no secret here. You just have to use the `Tasks.destroy()` function which uses as arguments an object containing data to find and delete a task. In our case, we'll use `req.params.id` to remove a single task and, as a successful response, uses `204 - No Content` as status code via `res.sendStatus(204)` function:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .put((req, res) => {
    Tasks.update(req.body, {where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .delete((req, res) => {
    Tasks.destroy({where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
```

So, we finish to *CRUDify* the task's resources of our API!

Refactoring some middlewares

To avoid code duplication, we'll apply a simple refactoring migrating the business logic used inside `app.all()` function to an Express middleware, using the `app.use()` function, into `libs/middlewares.js` file. To apply this refactoring, first, remove the functions `app.all()` from `routes/tasks.js` file, leaving the code with the following structure:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .get((req, res) => {
      // GET /tasks callback...
    })
    .post((req, res) => {
      // POST /tasks callback...
    });

  app.route("/tasks/:id")
    .get((req, res) => {
      // GET /tasks/id callback...
    })
    .put((req, res) => {
      // PUT /tasks/id callback...
    })
    .delete((req, res) => {
      // DELETE /tasks/id callback...
    });
};
```

To enable a JSON parse inside all API's routes, we must install the `body-parser` module, you do it running:

```
npm install body-parser@1.15.0 --save
```

After this installation, open and edit `libs/middlewares.js` file to insert the `bodyParser.json()` function as middleware, and in the end, include the middleware exclusion logic using the `app.use()` callback, following the code below:

```
import bodyParser from "body-parser";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
};
```

Creating user's endpoints

We also have to create routes to manage users in the application, after all, without them, it becomes impossible to manage tasks, am I right?

Our CRUD of users is not going to have anything new. In fact, won't be a complete CRUD, because we just need to create, find and delete an user, it won't be necessary to include the `app.route()` function too. Each route will be directly called by his correspondent HTTP method. The code will follow a similar treatment as task's routes has. The only new thing that will be used is the `attributes: ["id", "name", "email"]` inside the `Users.findById()` function. This parameter allows to return some selected fields from a model's result instead the full data of a model, it's similar to run the SQL query:

```
SELECT id, name, email FROM users;
```

To code the user's routes, creates the `routes/users.js` file, using the code below:

```
module.exports = app => {
  const Users = app.db.models.Users;

  app.get("/users/:id", (req, res) => {
    Users.findById(req.params.id, {
      attributes: ["id", "name", "email"]
    })
    .then(result => res.json(result))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  });

  app.delete("/users/:id", (req, res) => {
    Users.destroy({where: {id: req.params.id} })
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });

  app.post("/users", (req, res) => {
    Users.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
};
```

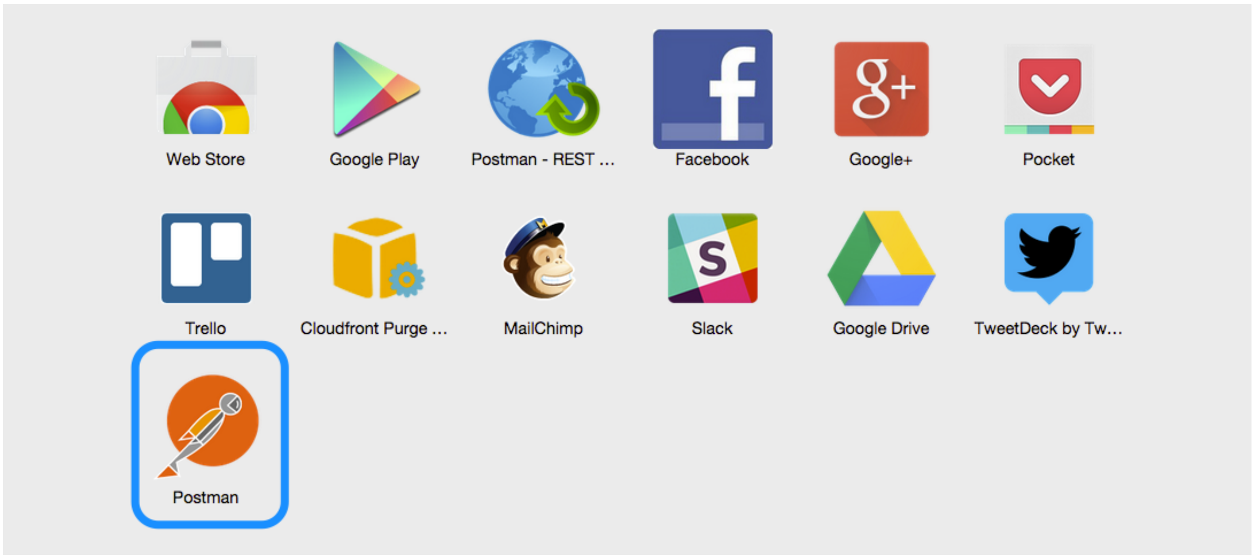
Attention: The reason why we're not using the function `app.route()` here is because in the next chapter, we are going to modify some specific points on each user's route, to find or delete only if the user is logged in the API.

Testing endpoint's access using Postman

To test these modifications, restart the application, open the browser and use some REST client application, because it will be necessary to test the `POST`, `PUT` and `DELETE` HTTP's methods. To simplify, I recommend you to use Postman, which is a useful Google Chrome extension and it's easy to use.

To install it go to: getpostman.com[9]

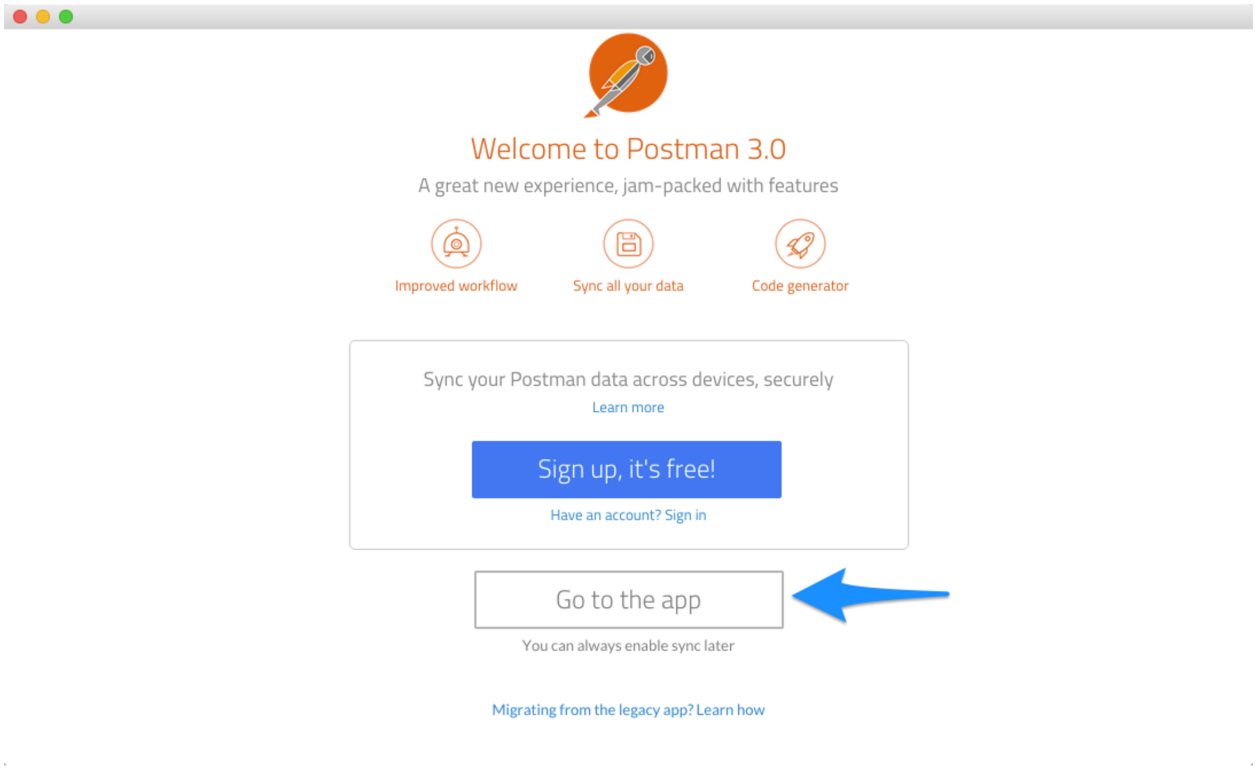
Then, after the installation, open the **Chrome Apps page** and click on the **Postman icon**.



Postman Rest Client app

A page to log in is going to be displayed, but you do not need to log in to use it. To skip this step and go straight to the main page, click on **Go to the app**:

[9]https://www.getpostman.com

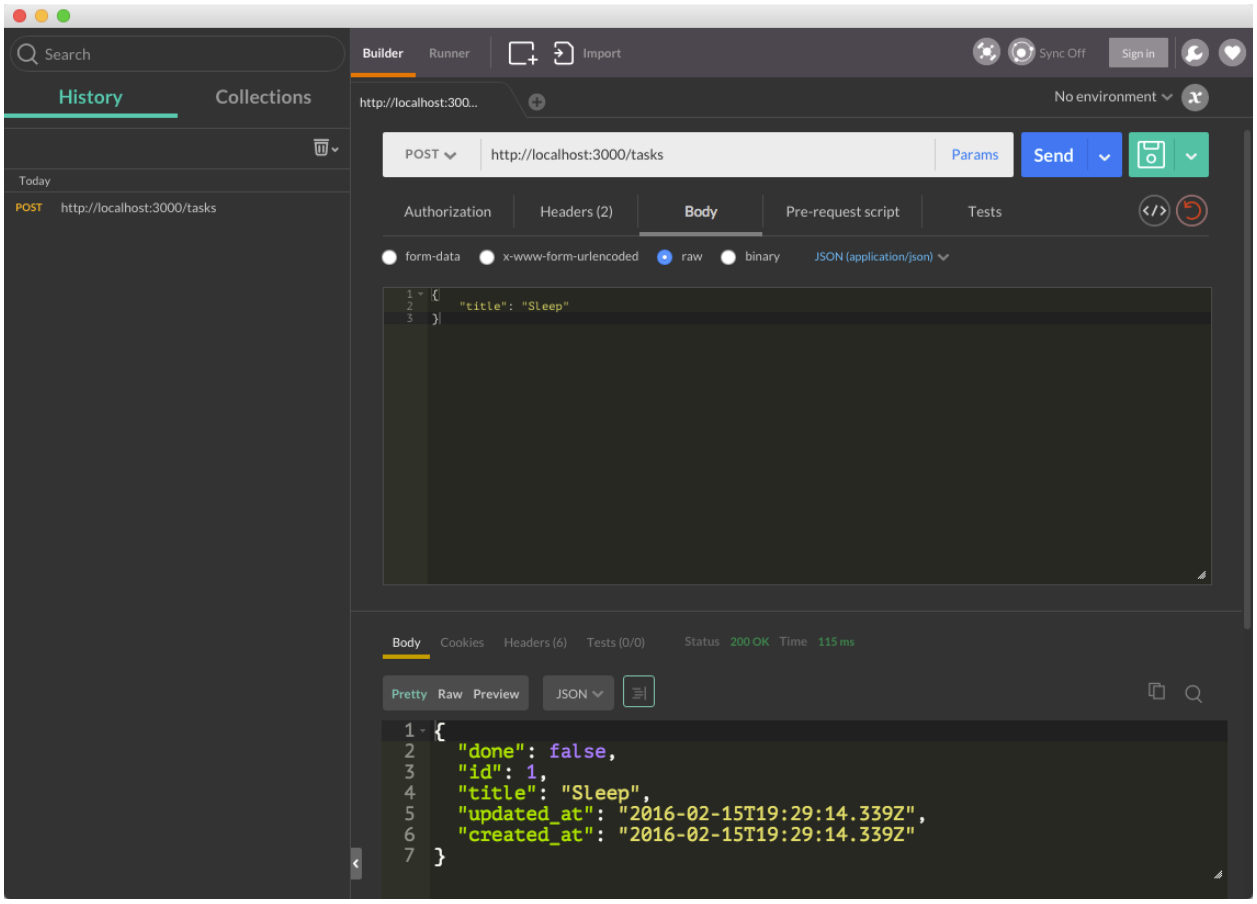

Welcome to Postman 3.0
A great new experience, jam-packed with features
Improved workflow
Sync all your data
Code generator
Sync your Postman data across devices, securely
Learn more
Sign up, it's free!
Have an account? Sign in
Go to the app
You can always enable sync later
Migrating from the legacy app? Learn how

Opening Postman

To test the endpoints, let's perform the following tests:

- Choose the method `POST` via address http://localhost:3000/tasks[10];
- Click on the menu **Body**, choose the option **raw**, and change the format from **Text** to **JSON (application/json)**;
- Create the JSON: `{"title": "Sleep"}` and click on the button **Send**;
- Modify the same JSON to `{"title": "Study"}` and click again on **Send**;

[10] http://localhost:3000/tasks

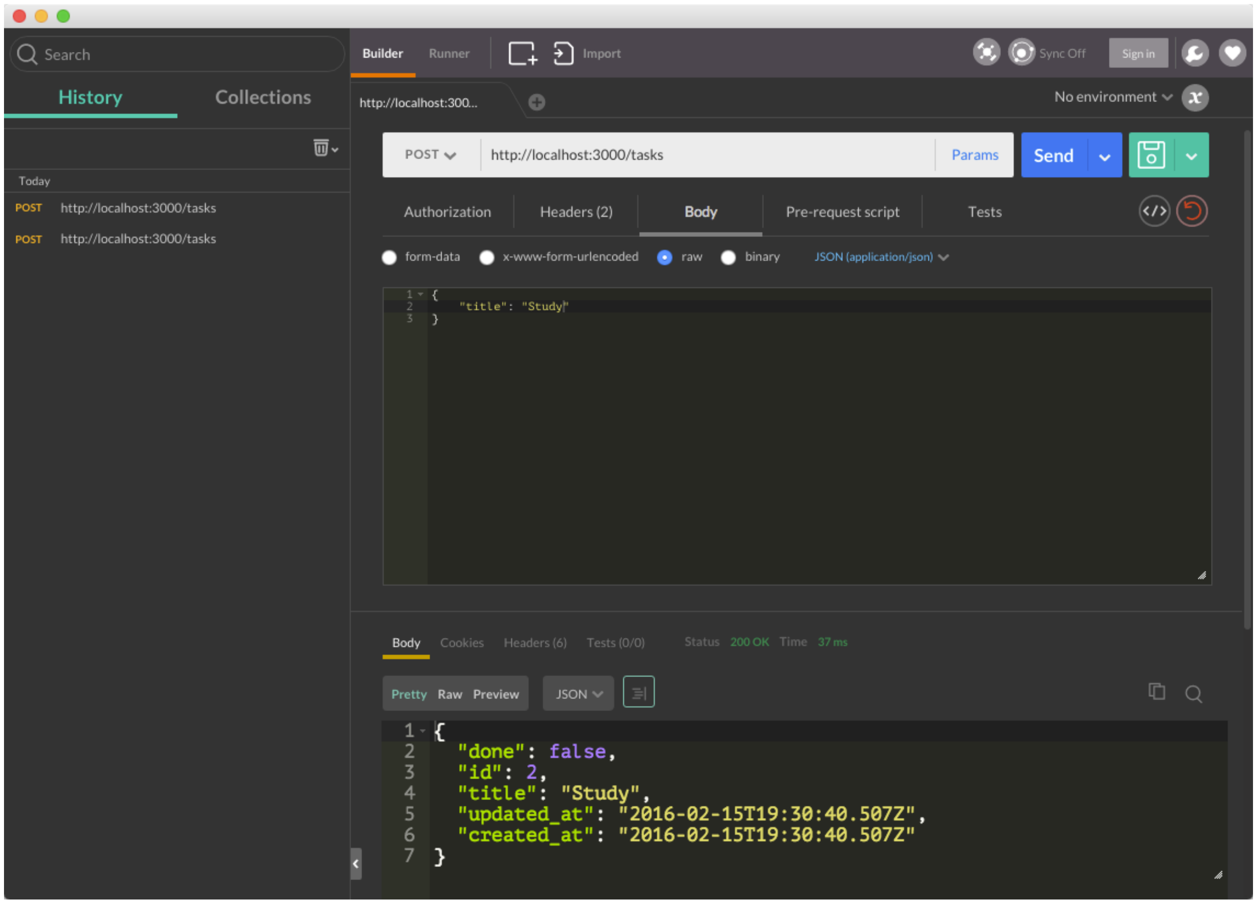


POST http://localhost:3000/tasks

```
{
    "title": "Sleep"
}
```

```
{
  "done": false,
  "id": 1,
  "title": "Sleep",
  "updated_at": "2016-02-15T19:29:14.339Z",
  "created_at": "2016-02-15T19:29:14.339Z"
}
```

Registering task: “Sleep”



POST http://localhost:3000/tasks

```
{
    "title": "Study"
}
```

Status 200 OK Time 37 ms

```
{
  "done": false,
  "id": 2,
  "title": "Study",
  "updated_at": "2016-02-15T19:30:40.507Z",
  "created_at": "2016-02-15T19:30:40.507Z"
}
```

Registering task: "Study"

With this procedure, you created two tasks, and now, you can explore the other task's routes. To test them, you can follow this simple list:

- Method `GET`, route http://localhost:3000/tasks[11];
- Method `GET`, route http://localhost:3000/tasks/1[12];
- Method `GET`, route http://localhost:3000/tasks/2[13];
- Method `PUT`, route http://localhost:3000/tasks/1[14] using the body: `{"title": "Work"}`;
- Method `DELETE`, route http://localhost:3000/tasks/2[15];

You can also test the user's endpoints, if you like.

[11]http://localhost:3000/tasks
[12]http://localhost:3000/tasks/1
[13]http://localhost:3000/tasks/2
[14]http://localhost:3000/tasks/1
[15]http://localhost:3000/tasks/2

Conclusion

Congratulations! If you reached at this point and alive, you have an outline of a RESTful API, which means, that now it's possible to build a client-side app to consume the tasks or users resources. Don't stop reading! Because in the next chapter we'll write some important things to make sure that our users will be authenticated by the API. See ya!

Authenticating users

Our API already has a tasks and users resources, which thanks to the Sequelize framework is integrated to a SQL database – in our case, SQLite3. We have already implemented their routes via the main routing functions provided by Express.

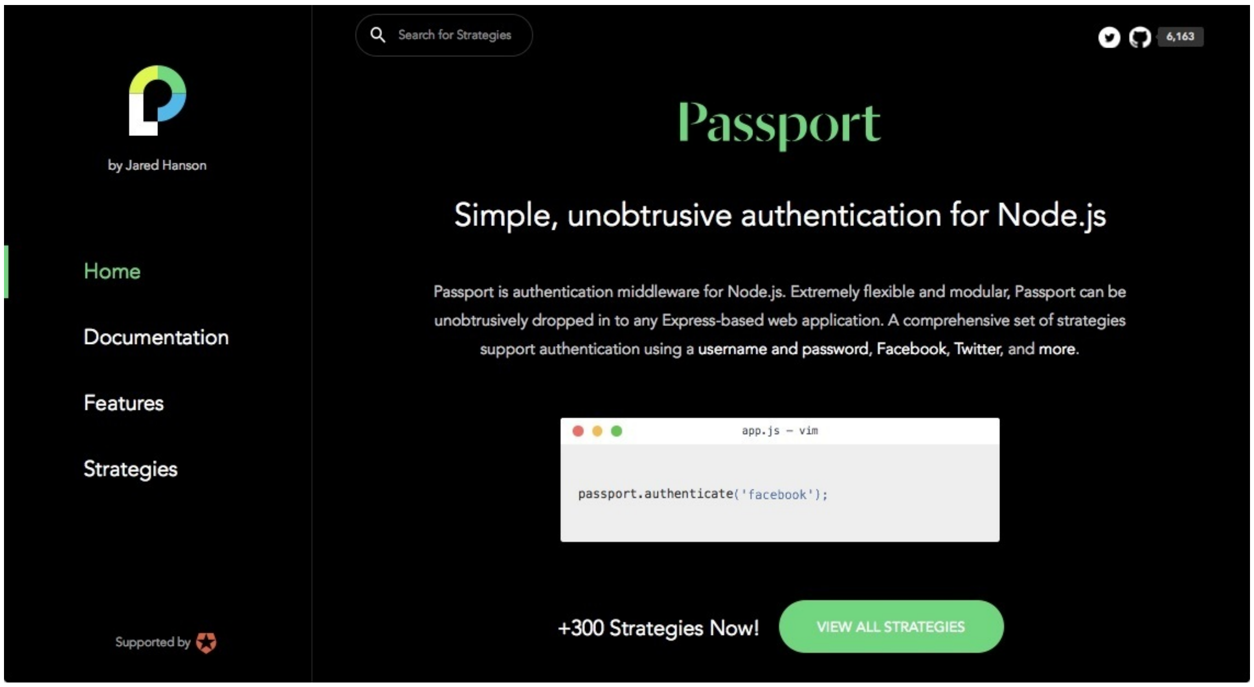
In this chapter, we'll explore the main concepts and implementations of user's authentication. After all, this is an important step to ensure that our users can manage their tasks safely.

Introduction to Passport.js and JWT

About Passport.js

There is a Node.js module very cool and easy to work with user's authentication, it's called **Passport**.

Passport is a framework that is extremely flexible and modular. It allows you to work with the main authentication strategies: **Basic & Digest**, **OpenID**, **OAuth**, **OAuth 2.0** and **JWT**. And also allows you to work with external services authentication, such as **Facebook**, **Google+**, **Twitter** and more. By the way, in its official website, **there is a list with +300 authentication strategies**, created and maintained by 3rd-party.



Simple, unobtrusive authentication for Node.js

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('facebook');
```

+300 Strategies Now!

Passport homepage

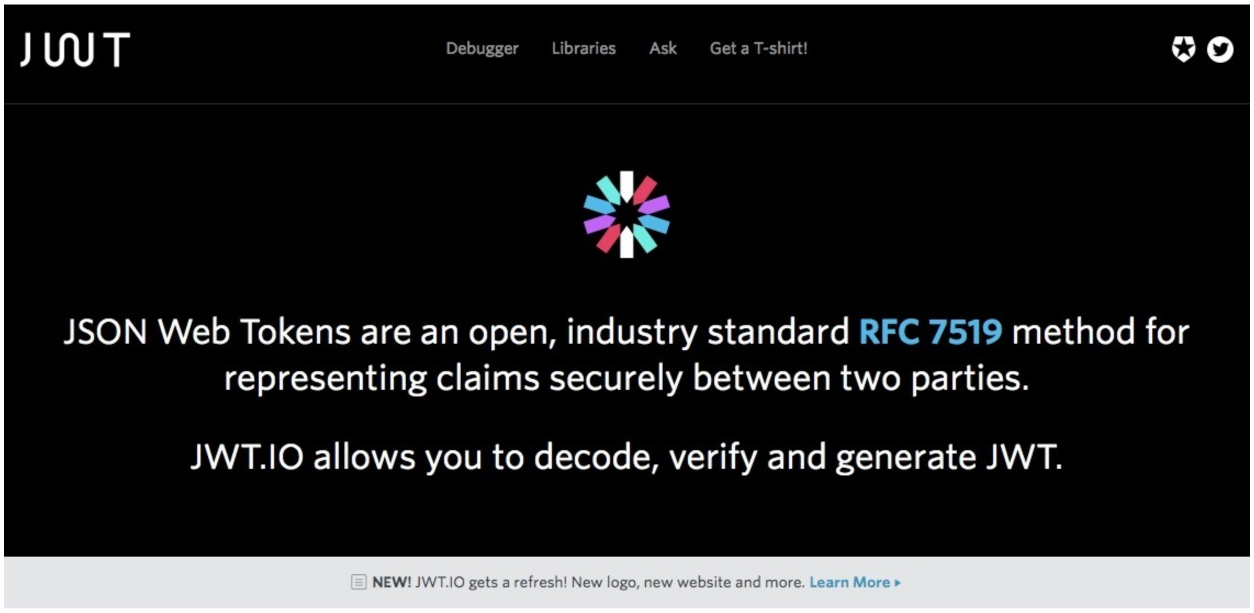
Its official website is: passportjs.org[16].

About JWT

JWT *(JSON Web Tokens)* is a very simple and secure authentication's strategy for REST APIs. It is a *open standard* for web authentications and is based in JSON token's requests between client and server. Its authentication's engine works like this:

1. Client makes a request once by sending their login credentials and password;
2. Server validates the credentials and, if everything is right, it returns to the client a JSON with token that encodes data from a user logged into the system;
3. Client, after receiving this token, can store it the way it wants, whether via LocalStorage, Cookie or other client-side storage mechanisms;
4. Every time the client accesses a route that requires authentication, it will only send this token to the API to authenticate and release consumption data; 5.Â Server always validate this token to allow or not a customer request.

To specific details about JWT, go to jwt.io[17].



JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.

JWT.IO allows you to decode, verify and generate JWT.

NEW! JWT.IO gets a refresh! New logo, new website and more. Learn More ▸

JWT homepage

[16]http://passportjs.org
[17]http://jwt.io

Installing Passport and JWT

To start the fun, we'll use the following modules:

- **Passport**: as authentication's engine;
- **Passport JWT**: as JWT authentication's strategy for Passport;
- **JWT Simple**: as encoder and decoder JSON tokens;

Now, let's install them running this command:

```
npm install passport@0.3.2 passport-jwt@2.0.0 jwt-simple@0.4.1 --save
```

To start this implementation, first we are going to add two new settings items for JWT (`jwtSecret` and `jwtSession`). Edit the `libs/config.js` file and in the end, add the following attributes:

```
module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    define: {
      underscored: true
    }
  },
  jwtSecret: "Nta$K-AP1",
  jwtSession: {session: false}
};
```

The field `jwtSecret` keeps a secret key string that serves as a base to **encode/decode** tokens. It's highly advisable to use a complex string which uses many different characters.

Never share or publish this secret key in public, because, if it leaks, you will let your application vulnerable, allowing a bad intentioned person to access the system and manages the tokens from logged users without informing the correctly credentials in the system.

To finish, the last included field is `jwtSession` that has the value `{session: false}`. This item is going to be used to inform Passport that the API won't manage session.

Implementing JWT authentication

Now that we have the Passport and JWT settings ready, let's implement the rules on how the client will be authenticated in our API. To start, we are going to implement the authentication rules, that will also have middleware functions provided by Passport to use into the API's routes. This code is going to have a middleware and two functions. The middleware will be executed in the moment it starts the application, and it basically receive in his callback a `payload` that contains a **decoded JSON** which was decoded using the secret key `cfg.jwtSecret`. This `payload` will have the attribute `id` that will be a user `id` to be used as argument for `Users.findById(payload.id)` function. As this middleware is going to be frequently accessed, to avoid some overheads, we are going to send a simple object containing only the `id` and `email` of the authenticated user, using the callback function:

```
done(null, {id: user.id, email: user.email});
```

This middleware will be injected via `passport.use(strategy)` function. To finish, two functions will be included from Passport to be used on the application. They are the `initialize()` function which starts the Passport and `authenticate()` which is used to authenticate the access for a route.

To understand better this implementation, let's create in the root folder the `auth.js` file, using this code:

```
import passport from "passport";
import {Strategy, ExtractJwt} from "passport-jwt";

module.exports = app => {
  const Users = app.db.models.Users;
  const cfg = app.libs.config;
  const params = {
    secretOrKey: cfg.jwtSecret,
    jwtFromRequest: ExtractJwt.fromAuthHeader()
  };
  const strategy = new Strategy(params, (payload, done) => {
      Users.findById(payload.id)
        .then(user => {
          if (user) {
            return done(null, {
              id: user.id,
              email: user.email
            });
          }
          return done(null, false);
        })
        .catch(error => done(error, null));
    });
  passport.use(strategy);
  return {
    initialize: () => {
      return passport.initialize();
    },
    authenticate: () => {
      return passport.authenticate("jwt", cfg.jwtSession);
    }
  };
};
```

To load the `auth.js` during the server boot time, edit the `index.js` code like this:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("libs/config.js")
  .then("db.js")
  .then("auth.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

To initiate the Passport middleware, edit the `libs/middlewares.js` and include the middleware `app.use(app.auth.initialize())`. See below were to include it:

```
import bodyParser from "body-parser";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
};
```

Generating tokens for authenticated users

To finish the JWT authentication, we are going to prepare the model `Users` to be able to encrypt the user's password. We also will create a route to generate tokens users who are going to authenticate themselves using their login and password on the system, and we'll do a refactoring in the tasks and users routes so that their access properly use the `id` of a authenticated user. Doing this, we finish this authentication step, making our application reliable and safer.

The encryption of user passwords will be performed by the module `bcrypt`. To do this, install it by running the command:

```
npm install bcrypt@0.8.5 --save
```

Now, let's edit the `Users` model including the function `hooks`, which executes functions before or after a database operation. In our case, will be included a function to be executed before registering a new user, via the function `beforeCreate()` to use the `bcrypt` to encrypt the user's password before save it.

A new function inside the `classMethods` will be included as well. It is going to be used to compare if the given password matches with the user's encrypted one. To encode these rules, edit the `models/users.js` with the following logic:

```
import bcrypt from "bcrypt";

module.exports = (sequelize, DataType) => {
  const Users = sequelize.define("Users", {
    // Users fields, defined in the chapter 5...
  }, {
    hooks: {
      beforeCreate: user => {
        const salt = bcrypt.genSaltSync();
        user.password = bcrypt.hashSync(user.password, salt);
      }
    },
    classMethods: {
      associate: models => {
        Users.hasMany(models.Tasks);
      },
      isPassword: (encodedPassword, password) => {
        return bcrypt.compareSync(password, encodedPassword);
      }
    }
  });
  return Users;
};
```

With these implemented modifications on the model `Users`, now we can create the endpoint `/token`. This route will be responsible for generating an encoded token with a `payload`, given to the user that sends the right e-mail and password via `req.body.email` e `req.body.password`.

The `payload` is going to have only the user `id`. The token generation occurs via `jwt-simple` module using the function `jwt.encode(payload, cfg.jwtSecret)` that mandatorily, will use the same secret key `jwtSecret` which was created on the `libs/config.js` file. Any error in this route will be treated using the HTTP `401 - Unauthorized` status code using the `res.sendStatus(401)` function.

To include this rule of tokens generation, you need to create the `routes/token.js` file using the following code:

```
import jwt from "jwt-simple";

module.exports = app => {
  const cfg = app.libs.config;
  const Users = app.db.models.Users;
  app.post("/token", (req, res) => {
    if (req.body.email && req.body.password) {
      const email = req.body.email;
      const password = req.body.password;
      Users.findOne({where: {email: email}})
        .then(user => {
          if (Users.isPassword(user.password, password)) {
            const payload = {id: user.id};
            res.json({
              token: jwt.encode(payload, cfg.jwtSecret)
            });
          } else {
            res.sendStatus(401);
          }
        })
        .catch(error => res.sendStatus(401));
    } else {
      res.sendStatus(401);
    }
  });
};
```

We already have the user's authentication and also the token generation's logic. To finish, let's use the `app.auth.authenticate()` function to validates the tokens sent by the clients and allows (or not) the access in some routes. To do this, edit the `routes/tasks.js` file and add the middleware function `all(app.auth.authenticate())` in the beginning of both endpoints. See below how will be:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .all(app.auth.authenticate())
    .get((req, res) => {
      // "/tasks": List tasks
    })
    .post((req, res) => {
      // "/tasks": Save new task
    });

  app.route("/tasks/:id")
    .all(app.auth.authenticate())
    .get((req, res) => {
      // "/tasks/1": Find a task
    })
    .put((req, res) => {
      // "/tasks/1": Update a task
    })
    .delete((req, res) => {
      // "/tasks/1": Delete a task
    });
};
```

When a client sends a valid token, their access will be successfully authenticated and consequently, the object `req.user` appears to be used inside the routes. This object is only created when the `auth.js` logic returns an authenticated user, that is, only when the following function returns a valid user:

```
// Do you remember this function from auth.js?
Users.findById(payload.id)
  .then(user => {
    if (user) {
      return done(null, {
        id: user.id,
        email: user.email
      });
    }
```

```
      return done(null, false);
    })
    .catch(error => done(error, null));
```

The `done()` callback send the authenticated user's data to the authenticated routes receive these data via `req.user` object. In our case, this object only has the `id` and `email` attributes.

To ensure the proper access to tasks resources, let's do a refactoring on all Sequelize functions from the routes `/tasks` and `/tasks/:id` to their queries use as parameter the `req.user.id`. To do this, edit the `routes/tasks.js` and, in the routes `app.route("/tasks")`, do the following modifications:

```
app.route("/tasks")
  .all(app.auth.authenticate())
  .get((req, res) => {
    Tasks.findAll({
      where: { user_id: req.user.id }
    })
    .then(result => res.json(result))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .post((req, res) => {
    req.body.user_id = req.user.id;
    Tasks.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
```

In the same file, do the same modification inside the queries of the `app.route("/tasks/:id")` function:

```
app.route("/tasks/:id")
  .all(app.auth.authenticate())
  .get((req, res) => {
    Tasks.findOne({ where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => {
      if (result) {
        return res.json(result);
      }
      return res.sendStatus(404);
    })
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .put((req, res) => {
    Tasks.update(req.body, { where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => res.sendStatus(204))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .delete((req, res) => {
    Tasks.destroy({ where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => res.sendStatus(204))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  });
```

To finish these refactoring, let's modify the users resources! Now we need to adapt some pieces of code inside the user's routes. Basically, we are going to change the way of how to search or delete an user to use the authenticated user's `id`.

In this case, won't be necessary to use an `id` in the route parameter, because now the route

`/users/:id` will be just `/user` (in the singular, because we are now dealing with a single logged user). Only to search and delete will have an authentication middleware, so, now both of them can be grouped via `app.route("/user")` function to reuse the middleware `all(app.auth.authenticate())`. Instead of `req.params.id`, we are going to use `req.user.id`, to ensure that an authenticated user `id` will be used.

To understand better how this logic will works, edit the `routes/users.js` file and do the following modifications:

```
module.exports = app => {
  const Users = app.db.models.Users;

  app.route("/user")
    .all(app.auth.authenticate())
    .get((req, res) => {
      Users.findById(req.user.id, {
        attributes: ["id", "name", "email"]
      })
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
    })
    .delete((req, res) => {
      Users.destroy({where: {id: req.user.id} })
        .then(result => res.sendStatus(204))
        .catch(error => {
          res.status(412).json({msg: error.message});
        });
    });

  app.post("/users", (req, res) => {
    Users.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
};
```

Conclusion

Congratulations! We have finished an extremely important step of the application. This time, all tasks will be returned only by an authenticated user. Thanks to JWT, now we have a safe mechanism for user’s authentication between client and server.

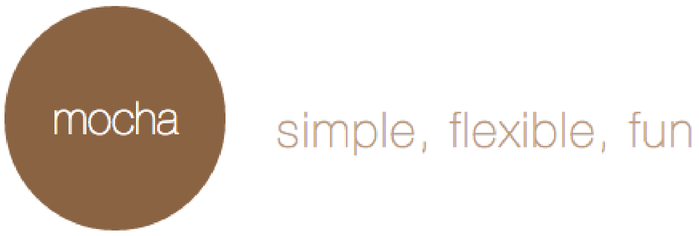
Up to now, it was implemented all the application back-end and we did not create a client application that uses all the power of our API. But chill out, there are a lots of surprises in the next chapters that are going to get you excited, just keep reading!

Testing the application - Part 1

Introduction to Mocha

Creating automated tests is something highly recommended to use. There are several types of tests: **unitary**, **functional**, **acceptance** and more. In this chapter, let's focus only in the **acceptance's test**, in our case aims to test the outputs and behaviors of our API's routes.

To create and execute the tests, it's necessary to use a test runner. We'll use the Mocha, which is a very popular in Node.js community.



Mocha test runner

Mocha has the following features:

- TDD style;
- BDD style;
- Code coverage HTML report;
- Customized test reports;
- Asynchronous test support;
- Easily integrated with the modules: `should`, `assert` and `chai`.

It is a complete environment to write tests and you learn more accessing his website: mochajs.org[18].

Setting up the test environment

To setup our test's environment, first we are going to setup a new database that is going to be used for us to play with **some fake data**. This practice is largely used to make sure that an application

[18]https://mochajs.org

can work easily in multiple environments. For now, our API has just single environment, because all the examples were developed in the `development` environment.

To enable the support for multiple environments, let's rename the current `libs/config.js` file to `libs/config.development.js` and then, create the `libs/config.test.js` file. The only new parameter in this new file is the `logging: false` which disables the SQL logs outputs. This will be necessary to disable these logs to not mess the tests report in the terminal. See how this file (`libs/config.test.js`) must look like:

```
module.exports = {
  database: "ntask_test",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    logging: false,
    define: {
      underscored: true
    }
  },
  jwtSecret: "NTASK_TEST",
  jwtSession: {session: false}
};
```

Now, we have to setup some files, each one has specific data to his correspondent environments. To load the settings according to the current environment, we must create a code to identifies what environment it is. In this case, we can use the `process.env` object which basically returns a lot of several environment variables from the OS.

A good practice in Node.js projects is to work with the variable `process.env.NODE_ENV` and use his value as a base, our application will have to load the settings from `test` or `development` environment. By default, the `development` will be defined when `process.env.NODE_ENV` returns null or an empty string.

Based on this brief explanation, let's recreate the `libs/config.js` file to load the settings according to the right system environment:

```
module.exports = app => {
  const env = process.env.NODE_ENV;
  if (env) {
    return require(`./config.${env}.js`);
  }
  return require("./config.development.js");
};
```

In our project, we are going to explore the **acceptance tests** only. To create them, we need to use these modules:

- `babel-register`: to run ES6 codes;
- `mocha`: to run the tests;
- `chai` to write BDD tests;
- `supertest` to do some requests in the API;

All these modules will be installed as a `devDependencies` in the `package.json` to use them only as test dependency. To do this, you need to use the `--save-dev` flag in the `npm install` command. See the command below:

```
npm install babel-register@6.5.2 mocha@2.4.5 chai@3.5.0 supertest@1.2.0 --save-d\
ev
```

Now, let's encapsulate the `mocha` test runner into the `npm test` alias command, to internally run the command: `NODE_ENV=test mocha test/**/*.js`. To implement this new command, edit the `package.json` and include the `scripts.test` attribute:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "Task list API",
  "main": "index.js",
  "scripts": {
    "start": "babel-node index.js",
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "bcrypt": "^0.8.5",
    "body-parser": "^1.15.0",
    "consign": "^0.1.2",
    "express": "^4.13.4",
    "jwt-simple": "^0.4.1",
    "passport": "^0.3.2",
    "passport-jwt": "^2.0.0",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1"
  },
  "devDependencies": {
    "babel-register": "^6.5.2",
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

Then, we are going to export our main API module, the `index.js`, to allow the API can be started during the tests. To do it, you must include the `module.exports = app` in the end of the `index.js` file and we'll also disable the logs created by the `consign` module via `consign({verbose: false})` settings to not pollute the tests report.

```
import express from "express";
import consign from "consign";

const app = express();

consign({verbose: false})
  .include("libs/config.js")
  .then("db.js")
  .then("auth.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);

module.exports = app;
```

Now, the application can be internally started by the `supertest` module during the tests. To avoid the server runs twice in the tests environment, you need to modify the `libs/boot.js` to run the database sync and server listen only when `process.env.NODE_ENV` has not the `test` value.

To change this, open and edit the `libs/boot.js` using the simple code below:

```
module.exports = app => {
  if (process.env.NODE_ENV !== "test") {
    app.db.sequelize.sync().done(() => {
      app.listen(app.get("port"), () => {
        console.log(`NTask API - Port ${app.get("port")}`);
      });
    });
  }
};
```

To finish our test's environment setup, let's prepare some Mocha specific settings, to load the API server and the modules `chai` and `supertest` as global variables. This will accelerate the execution of tests, after all, each one will load these modules again and again, and if we put all main things to load once, will save some milliseconds of the tests execution. To implement this simple practice, create the file `test/helpers.js`:

```
import supertest from "supertest";
import chai from "chai";
import app from "../index.js";

global.app = app;
global.request = supertest(app);
global.expect = chai.expect;
```

Then, let's create a simple file which allows to include some settings as parameters to the `mocha` command. This will be responsible to load the `test/helpers.js` and also use the `--reporter spec` flag to show a detailed report about the tests. After that, we'll include the `--compilers js:babel-register` flag, to Mocha be able to run the tests in **ECMAScript 6** standard via `babel-register` module.

The last flag is `--slow 5000`, this will wait five seconds before start all tests (time enough to start the API server and database connection safely). Create the `test/mocha.opts` file using the following parameters:

```
--require test/helpers
--reporter spec
--compilers js:babel-register
--slow 5000
```

Writing the firsts tests

We finished the setup about the test environment. What about test something? What about write some tests code for `routes/index.js`? It's very simple to test: basically we will make sure the

API is returning the JSON correctly, comparing the results with the static JSON `const expected = {status: "NTask API"}` to see if both are matching.

To create our first test, let's use the `request.get("/")` function, to validate if this request is returning the status `200`. To finish this test, we check if the `req.body` and `expected` are the same using `expect(res.body).to.eql(expected)` function.

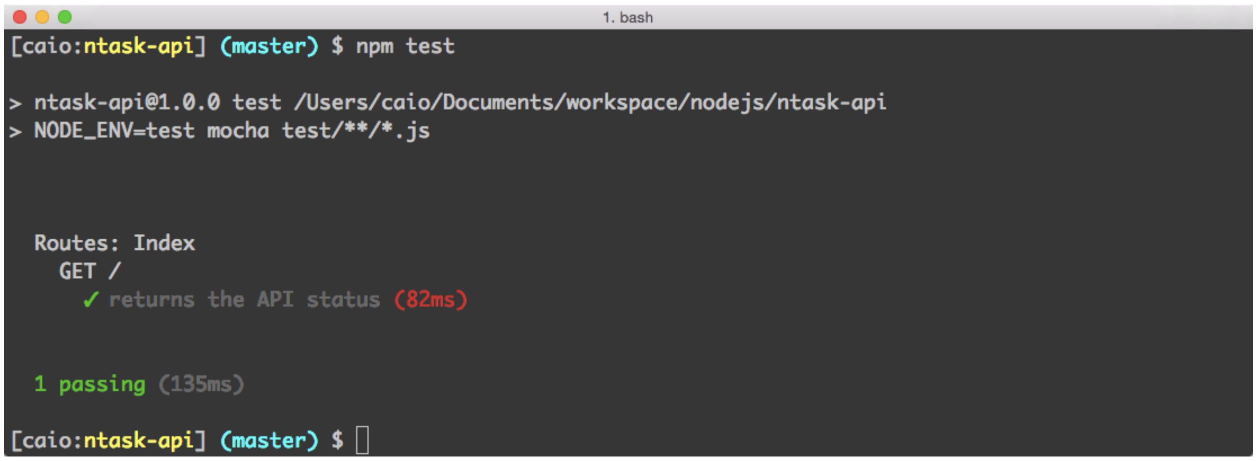
To implement this test, create the `test/routes/index.js` file using the following codes:

```
describe("Routes: Index", () => {
  describe("GET /", () => {
    it("returns the API status", done => {
      request.get("/")
        .expect(200)
        .end((err, res) => {
          const expected = {status: "NTask API"};
          expect(res.body).to.eql(expected);
          done(err);
        });
    });
  });
});
```

To execute this test, run the command:

```
npm test
```

After the execution, you will have a similar output like this image:



```
[caio:ntask-api] (master) $ npm test

> ntask-api@1.0.0 test /Users/caio/Documents/workspace/nodejs/ntask-api
> NODE_ENV=test mocha test/**/*.js



  Routes: Index
    GET /
      ✓ returns the API status (82ms)


  1 passing (135ms)

[caio:ntask-api] (master) $
```

Running the first test

Testing the authentication endpoint

In this section, we'll implement several tests. To start off, let's test the endpoint from `routes/token.js`, which is responsible to generates JSON web tokens for authenticated users.

Basically, this endpoint will have four tests to validate:

- Request authenticated by a valid user;
- Request with a valid e-mail but with wrong password;
- Request with an unregistered e-mail;
- Request without email and password;

Create the test `test/routes/token.js` with the following structure:

```
describe("Routes: Token", () => {
  const Users = app.db.models.Users;
  describe("POST /token", () => {
    beforeEach(done => {
      // Runs before each test...
    });
    describe("status 200", () => {
      it("returns authenticated user token", done => {
        // Test's logic...
      });
    });
    describe("status 401", () => {
      it("throws error when password is incorrect", done => {
        // Test's logic...
      });
      it("throws error when email not exist", done => {
        // Test's logic...
      });
      it("throws error when email and password are blank", done => {
        // Test's logic...
      });
    });
  });
});
```

To start write these tests, first, we're gonna code some queries to clear the user's table and create one valid user inside `beforeEach()` callback. This function will be executed before each test. To do this, we'll use the model `app.db.models.Users` and his functions: `Users.destroy({where: {}})` to clean the user's table and `Users.create()` to save a single valid user for each test execution. This will allow to test the main flows of this route:

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
      password: "12345"
    }))
    .then(done());
});
```

Now, we are going to implement test by test. The first test is a successful case. To test it, let's use the function `request.post("/token")` to request a token by sending the email and the password of a valid user via `send()` function.

To finish the test, the `end(err, res)` callback must return `res.body` with the `token` key to be checked via `expect(res.body).to.include.keys("token")` function. To conclude a test, it's required to execute the callback `done()` in the end of a test.

Always send the variable `err` as a parameter into `done(err)` function, because if something wrong happens during the tests, this function will show the details of the error. See below the complete code of this first test:

```
it("returns authenticated user token", done => {
  request.post("/token")
    .send({
      email: "john@mail.net",
      password: "12345"
    })
    .expect(200)
    .end((err, res) => {
      expect(res.body).to.include.keys("token");
      done(err);
    });
});
```

After this first test, we'll write some tests to verify if the errors is happening well. Now, let's test the request of an invalid password, expecting a `401 unauthorized access` status code. This tests is simpler, because basically we'll test if the request returns the `status 401` error, via the function `expect(401)`:

```
it("throws error when password is incorrect", done => {
  request.post("/token")
    .send({
      email: "john@mail.net",
      password: "WRONG_PASSWORD"
    })
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

The next test is very similar to the last one, but now, it'll be tested the invalid user's email behavior, expecting the request returns `401` status code again:

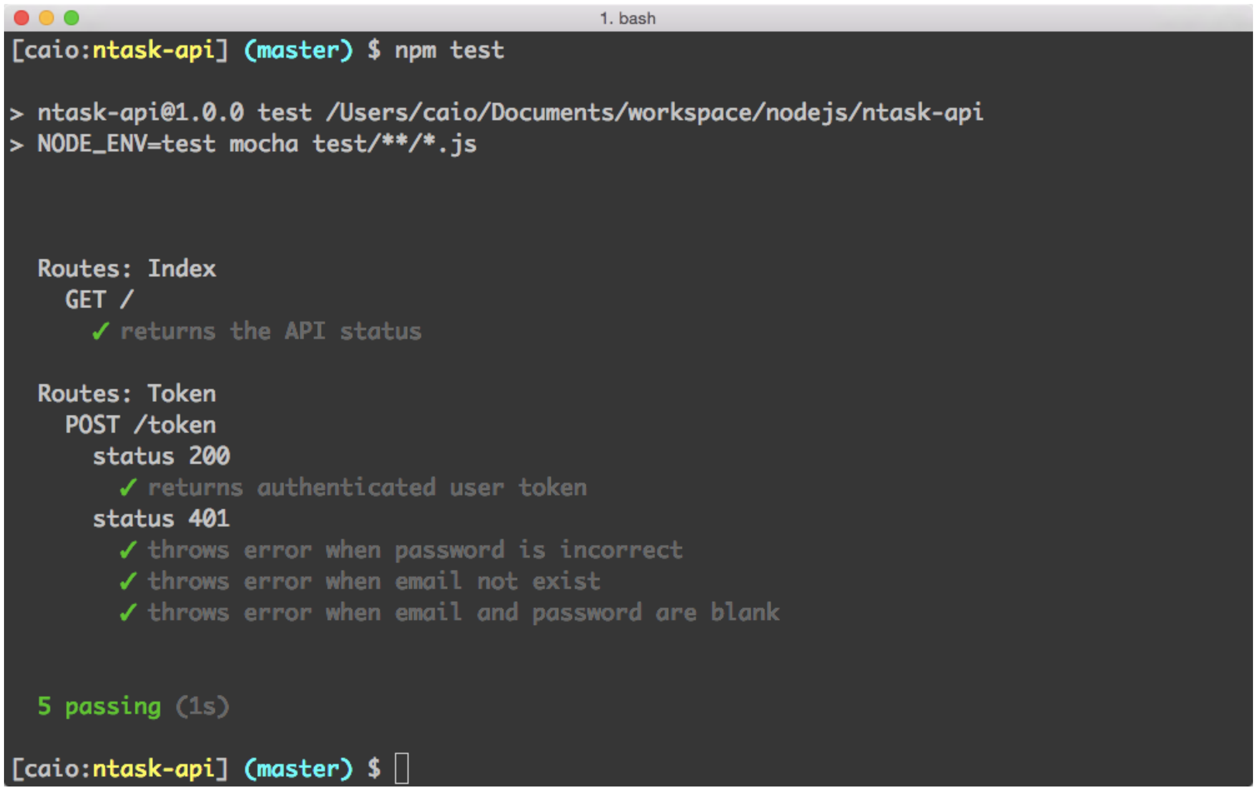
```
it("throws error when email not exist", done => {
  request.post("/token")
    .send({
      email: "wrong@email.com",
      password: "12345"
    })
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

And, to finish this test case, let's check if we get the same `401` status code when no email and no password are sent. This one is even simpler, because we don't need to send parameters in this request.

```
it("throws error when email and password are blank", done => {
  request.post("/token")
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

Conclusion

Congrats! To run all tests, just type the `npm test` command again, now, probably you'll get a similar result as this image is:



```
[caio:ntask-api] (master) $ npm test

> ntask-api@1.0.0 test /Users/caio/Documents/workspace/nodejs/ntask-api
> NODE_ENV=test mocha test/**/*.js



  Routes: Index
    GET /
      ✓ returns the API status

  Routes: Token
    POST /token
      status 200
        ✓ returns authenticated user token
      status 401
        ✓ throws error when password is incorrect
        ✓ throws error when email not exist
        ✓ throws error when email and password are blank


  5 passing (1s)

[caio:ntask-api] (master) $
```

Test's result

Keep reading because the tests subject is long and we'll keep talking about it in the next chapter to write more tests for our API's routes.

Testing the application - Part 2

Continuing the tests implementation, now we will focus on write some tests for the resources: **tasks** and **users**.

Testing task's endpoints

To test the endpoints of task's resource, we are going to cheat the **JWT authentication**. After all, will be necessary to correctly test the results of this resource and also the others resources which involves user's authentication. To start it, let's create the structure for the tasks tests.

Create the file `test/routes/tasks.js` with the following code:

```
import jwt from "jwt-simple";

describe("Routes: Tasks", () => {
  const Users = app.db.models.Users;
  const Tasks = app.db.models.Tasks;
  const jwtSecret = app.libs.config.jwtSecret;
  let token;
  let fakeTask;
  beforeEach(done => {
    // Runs before each test...
  });
  describe("GET /tasks", () => {
    describe("status 200", () => {
      it("returns a list of tasks", done => {
        // Test's logic...
      });
    });
  });
  describe("POST /tasks/", () => {
    describe("status 200", () => {
      it("creates a new task", done => {
        // Test's logic...
      });
    });
  });
  describe("GET /tasks/:id", () => {
    describe("status 200", () => {
      it("returns one task", done => {
        // Test's logic...
      });
    });
    describe("status 404", () => {
      it("throws error when task not exist", done => {
        // Test's logic...
      });
    });
  });
  describe("PUT /tasks/:id", () => {
    describe("status 204", () => {
      it("updates a task", done => {
        // Test's logic...
      });
    });
  });
  describe("DELETE /tasks/:id", () => {
    describe("status 204", () => {
      it("removes a task", done => {
        // Test's logic...
      });
    });
  });
});
```

Detailing how to cheat the authentication part, we are going to reuse the module `jwt-simple` to create a valid token which will be used in the header of all the tests. This token will be repeatedly generated within the callback of the function `beforeEach(done)`. But, to generate it, we have to delete all users first using the function `Users.destroy({where: {}})` and then, create a new one via `Users.create()` function.

We'll do the same with tasks creation, but instead of using the function `Tasks.create()` it will be used the function `Tasks.bulkCreate()` which allows sending an array of tasks to be inserted in a single execution (this function is very useful for inclusion in a plot of data).

The tasks are going to use the `user.id` field, created to ensure that they are from the authenticated user. In the end, let's use the first task created by the piece `fakeTask = tasks[0]` to reuse its `id` on the tests that need a task `id` as a route parameter. We'll generate a valid token using the function `jwt.encode({id: user.id}, jwtSecret)`.

Both the objects `fakeTask` and `token` are created in a scope above the function `beforeEach(done)`, so they can be reused on the tests. To understand in detail, you need to write the following implementation:

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
      password: "12345"
    }))
    .then(user => {
      Tasks
        .destroy({where: {}})
        .then(() => Tasks.bulkCreate([{
          id: 1,
          title: "Work",
          user_id: user.id
        }, {
          id: 2,
          title: "Study",
          user_id: user.id
        }]))
        .then(tasks => {
          fakeTask = tasks[0];
          token = jwt.encode({id: user.id}, jwtSecret);
          done();
        });
    });
});
```

With the pre-test routine ready, we are going to write all the tasks tests, starting with the `GET /` route. On it, it is performed a request via `request.get("/tasks")` function, using also the function `set("Authorization", ` JWT ${token}) to allows sending a header on the request, which in this case, the header `Authorization` is sent along with the value of a valid token.

To make sure the test is successfully accomplished:

1. We check the `status 200` via `expect(200)` function;
2. We apply a simple validation to ensure that the array of size `2` will be returned via `expect(res.body).to.have.length 2)` function;
3. We compare if the titles of the first two tasks are the same as those created by the function `Tasks.bulkCreate()`;

```
describe("GET /tasks", () => {
  describe("status 200", () => {
    it("returns a list of tasks", done => {
      request.get("/tasks")
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body).to.have.length(2);
          expect(res.body[0].title).to.eql("Work");
          expect(res.body[1].title).to.eql("Study");
          done(err);
        });
    });
  });
});
```

To test the successful case of the route `POST /tasks`, there is no secret: basically is informed a header with an authentication token and a title for a new task. As result, we test if the answer returns `200` status code and if the object `req.body` has the same title as the one which was sent to register a new task.

```
describe("POST /tasks", () => {
  describe("status 200", () => {
    it("creates a new task", done => {
      request.post("/tasks")
        .set("Authorization", `JWT ${token}`)
        .send({title: "Run"})
        .expect(200)
        .end((err, res) => {
          expect(res.body.title).to.eql("Run");
          expect(res.body.done).to.be.false;
          done(err);
        });
    });
  });
});
```

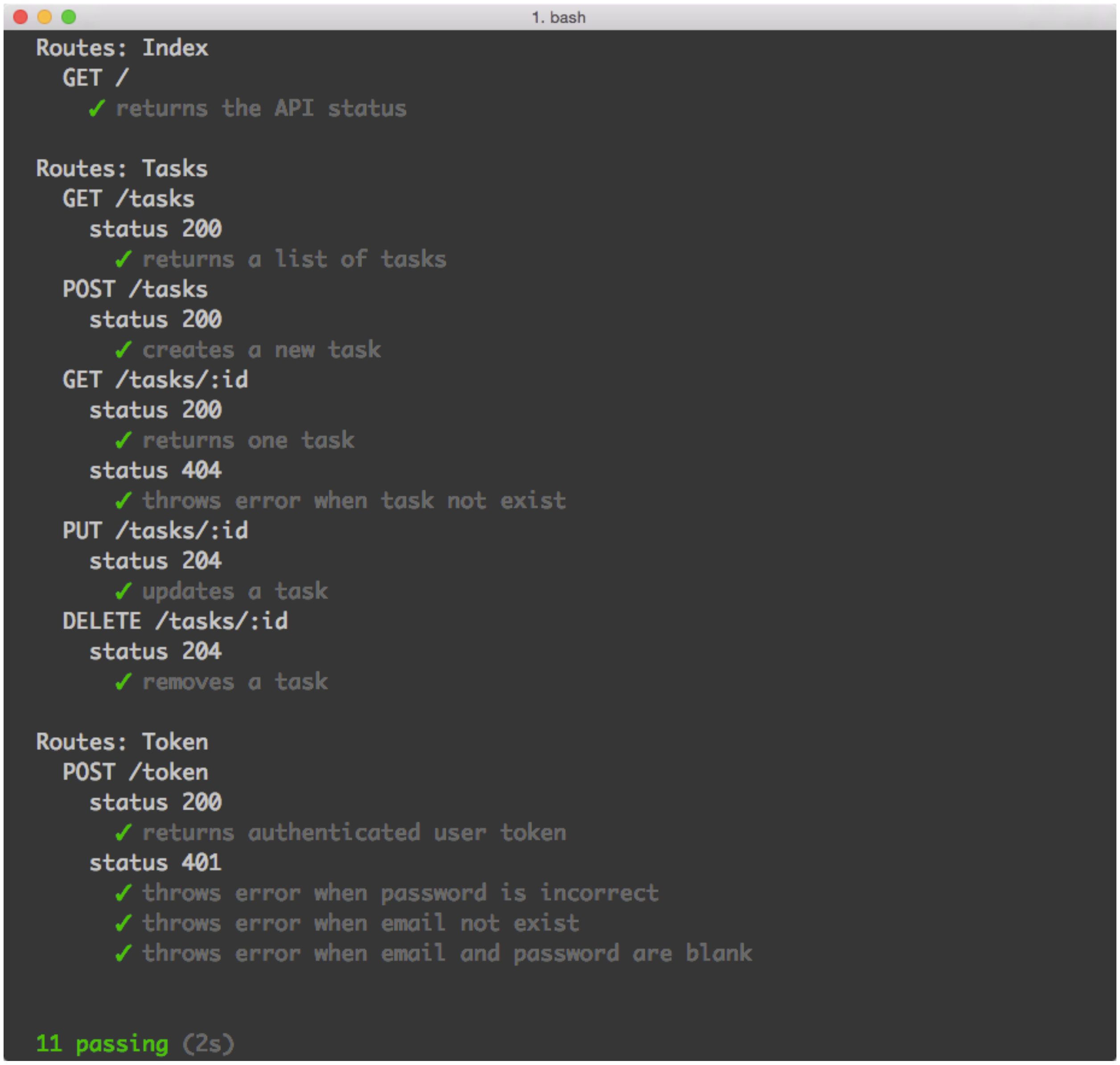
Now we are going to test two simple flows of the route `GET /tasks/:id`. In the successful case we'll use the `id` of the object `fakeTask` to make sure a valid task will be returned. To test how the application behaves when a `id` of an invalid task is informed, we are going to use the function `expect(404)` to test the `status 404` that indicates if the request did not find a resource.

```
describe("GET /tasks/:id", () => {
  describe("status 200", () => {
    it("returns one task", done => {
      request.get(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body.title).to.eql("Work");
          done(err);
        });
    });
  });
  describe("status 404", () => {
    it("throws error when task not exist", done => {
      request.get("/tasks/0")
        .set("Authorization", `JWT ${token}`)
        .expect(404)
        .end((err, res) => done(err));
    });
  });
});
```

To finish the tests, we are going to test the successful behavior of the routes `PUT /tasks/:id` and `DELETE /tasks/:id`. Both of them will use the same functions, except that a test is going to execute the function `request.put()` and the other one is going to execute `request.delete()`. But, both of them expect that the request returns a `204` status code via `expect(204)` function.

```
describe("PUT /tasks/:id", () => {
  describe("status 204", () => {
    it("updates a task", done => {
      request.put(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
        .send({
          title: "Travel",
          done: true
        })
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
describe("DELETE /tasks/:id", () => {
  describe("status 204", () => {
    it("removes a task", done => {
      request.delete(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
```

We have finished the tests of tasks resources. In case you execute the command `npm test` again, you are going to see the following result:



```
Routes: Index
  GET /
    ✓ returns the API status

Routes: Tasks
  GET /tasks
    status 200
      ✓ returns a list of tasks
  POST /tasks
    status 200
      ✓ creates a new task
  GET /tasks/:id
    status 200
      ✓ returns one task
    status 404
      ✓ throws error when task not exist
  PUT /tasks/:id
    status 204
      ✓ updates a task
  DELETE /tasks/:id
    status 204
      ✓ removes a task

Routes: Token
  POST /token
    status 200
      ✓ returns authenticated user token
    status 401
      ✓ throws error when password is incorrect
      ✓ throws error when email not exist
      ✓ throws error when email and password are blank


11 passing (2s)
```

Testing task's resource

Testing user's endpoints

To test the user's resource is even simpler, because basically we are going to use all that were explained in the last tests. To start it, create the file `test/routes/users.js` with the following structure:

```
import jwt from "jwt-simple";

describe("Routes: Tasks", () => {
  const Users = app.db.models.Users;
  const jwtSecret = app.libs.config.jwtSecret;
  let token;
  beforeEach(done => {
    // Runs before each test...
  });
  describe("GET /user", () => {
    describe("status 200", () => {
      it("returns an authenticated user", done => {
        // Test's logic...
      });
    });
  });
  describe("DELETE /user", () => {
    describe("status 204", () => {
      it("deletes an authenticated user", done => {
        // Test's logic...
      });
    });
  });
  describe("POST /users", () => {
    describe("status 200", () => {
      it("creates a new user", done => {
        // Test's logic...
      });
    });
  });
});
```

The pre-test logic is going to be simplified, however it will have the generation of a valid token. See below how to implement the `beforeEach(done)` function:

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
      password: "12345"
    }))
    .then(user => {
      token = jwt.encode({id: user.id}, jwtSecret);
      done();
    });
});
```

Now, to implement these tests, let's get starting to test the `GET /user` route, which must returns the authenticated user's data which basically sends a token and receive as a response the user's data that were created from `beforeEach(done)` function.

```
describe("GET /user", () => {
  describe("status 200", () => {
    it("returns an authenticated user", done => {
      request.get("/user")
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body.name).to.eql("John");
          expect(res.body.email).to.eql("john@mail.net");
          done(err);
        });
    });
  });
});
```

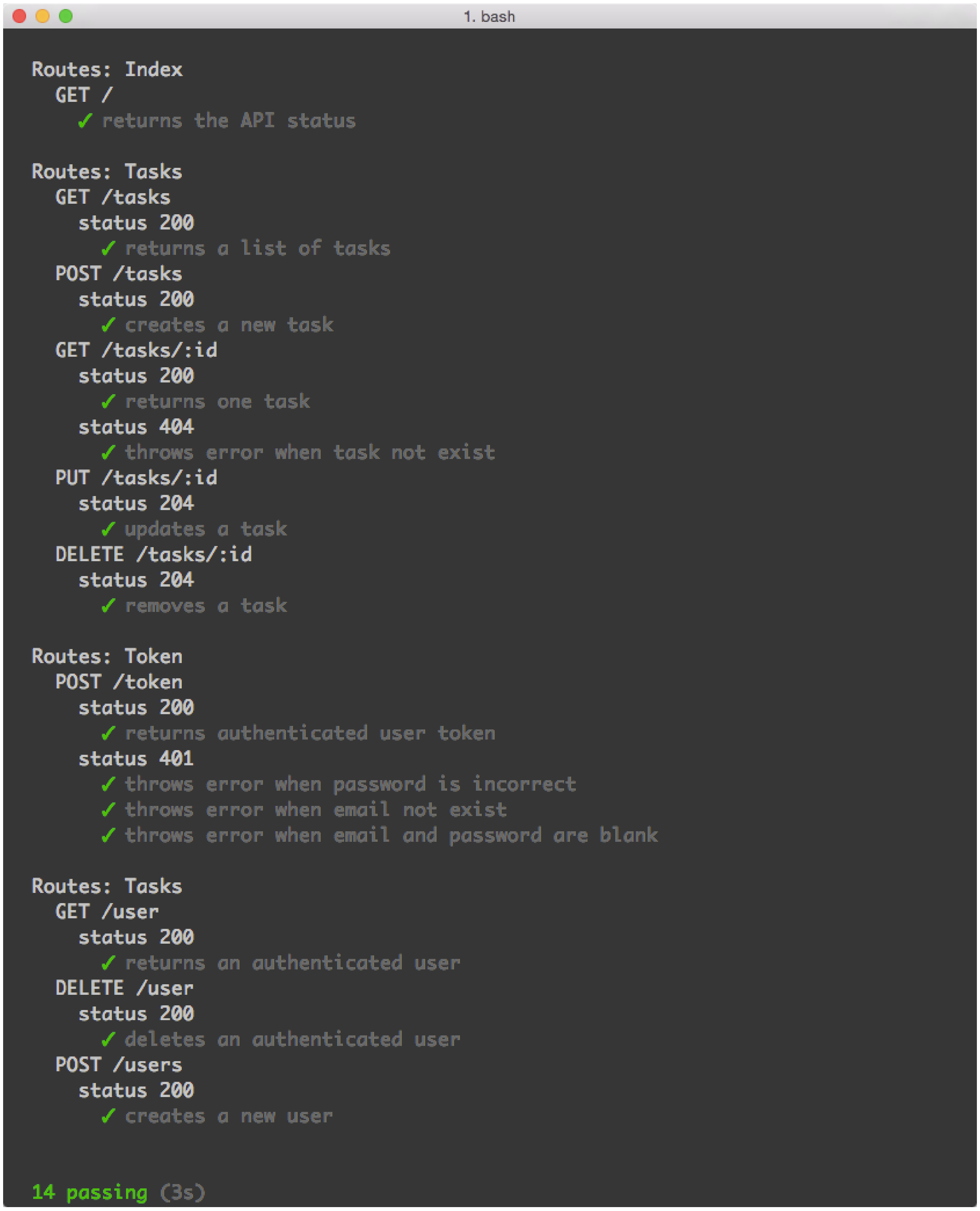
Then, let's write the tests to the `DELETE /user` route. In this case, is very simple, you just need to send a token and wait for the `204` status code.

```
describe("DELETE /user", () => {
  describe("status 204", () => {
    it("deletes an authenticated user", done => {
      request.delete("/user")
        .set("Authorization", `JWT ${token}`)
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
```

To finish the last test, we are going to implement the test for new user's route. This one, doesn't require a token, after all, it is an open route for new users register an account in the API. See below this test's code:

```
describe("POST /users", () => {
  describe("status 200", () => {
    it("creates a new user", done => {
      request.post("/users")
        .send({
          name: "Mary",
          email: "mary@mail.net",
          password: "12345"
        })
        .expect(200)
        .end((err, res) => {
          expect(res.body.name).to.eql("Mary");
          expect(res.body.email).to.eql("mary@mail.net");
          done(err);
        });
    });
  });
});
```

Now, if you execute the command `npm test` again, you'll see a report like this:



```
Routes: Index
  GET /
    ✓ returns the API status

Routes: Tasks
  GET /tasks
    status 200
      ✓ returns a list of tasks
  POST /tasks
    status 200
      ✓ creates a new task
  GET /tasks/:id
    status 200
      ✓ returns one task
    status 404
      ✓ throws error when task not exist
  PUT /tasks/:id
    status 204
      ✓ updates a task
  DELETE /tasks/:id
    status 204
      ✓ removes a task

Routes: Token
  POST /token
    status 200
      ✓ returns authenticated user token
    status 401
      ✓ throws error when password is incorrect
      ✓ throws error when email not exist
      ✓ throws error when email and password are blank

Routes: Tasks
  GET /user
    status 200
      ✓ returns an authenticated user
  DELETE /user
    status 200
      ✓ deletes an authenticated user
  POST /users
    status 200
      ✓ creates a new user


14 passing (3s)
```

Testing user's resource

Conclusion

If you reached this step, then you have developed a small but powerful API using Node.js and SQL database. Everything is already working and was tested to ensure the code quality in the project.

In the next chapter, we'll use a very useful tool for generating API documentation. Keep reading because there are a lot of cool stuff to explore!

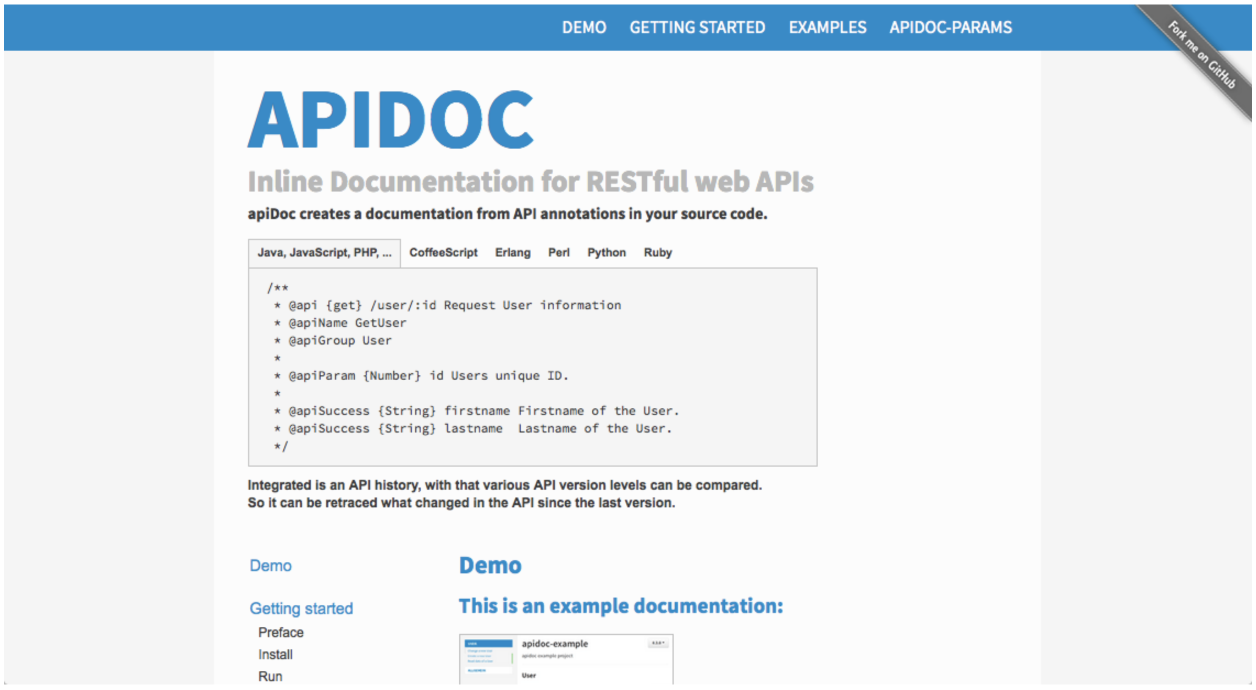
Documenting the API

If you reached this chapter and your application is working correctly – with routes to the management tasks and users, integrated to a database and with user's authentication via JSON Web Token – congratulations! You have created, following some best practices, an REST API using Node.js. If you intend to use this pilot project as a base to construct your own API, then you already have an application enough to deploy it into a production environment.

Introduction to ApiDoc.js

In this chapter, we'll learn how to write and generate a pretty API documentation, after all, it is a good practice to provide documentation about how the client applications can connect to consume the data from an API. The coolest thing is that we are going to use a very simple tool and all the documentation of our application will be built using code's comment.

Our project will use the **ApiDoc.js**, which is a Node.js module to generate elegants documentation for APIs.



apiDoc creates a documentation from API annotations in your source code.

```
/**
 * @api {get} /user/:id Request User information
 * @apiName GetUser
 * @apiGroup User
 *
 * @apiParam {Number} id Users unique ID.
 *
 * @apiSuccess {String} firstname Firstname of the User.
 * @apiSuccess {String} lastname  Lastname of the User.
 */
```

Integrated is an API history, with that various API version levels can be compared.
So it can be retraced what changed in the API since the last version.

Demo

This is an example documentation:

ApiDoc.js homepage

This module is a CLI *(Command Line Interface)* and it is highly advisable to install it as a global module (using the command `npm install -g`). However, in our case, we can use it as local module and we'll create a npm alias command to use every time we start the API server. So, its installation is going to be as a local module, similar to the others. Install it using the command:

```
npm install apidoc@0.15.1 --save-dev
```

First, let's create the command: `npm run apidoc` to execute internally the command `apidoc -i routes/ -o public/apidoc`. Then, modify the attribute `scripts.start` so it can generate the API documentation before start the API server. We also gonna include the attribute `apidoc.name` to set a title and the `apidoc.template.forceLanguage` to set the default language to be used in the API documentation.

Open and edit the `package.json`, doing these changes:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "Task list API",
  "main": "index.js",
  "scripts": {
    "start": "npm run apidoc && babel-node index.js",
    "apidoc": "apidoc -i routes/ -o public/apidoc",
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
  "apidoc": {
    "name": "Node Task API - Documentation",
    "template": {
      "forceLanguage": "en"
    }
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "bcrypt": "^0.8.5",
    "body-parser": "^1.15.0",
    "consign": "^0.1.2",
    "express": "^4.13.4",
    "jwt-simple": "^0.4.1",
    "passport": "^0.3.2",
    "passport-jwt": "^2.0.0",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1"
  },
  "devDependencies": {
    "apidoc": "^0.15.1",
    "babel-register": "^6.5.2",
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

Now, every time you run the command `npm start`, if you want to generate a new documentation without initiating the server, you just run `npm run apidoc` command. Both of the commands are going to search all the existent comments in the `routes` directory to generate a new API documentation, which will be published in the `public/apidoc` folder and then, start the server.

To be able to view the documentation page, first we have to enable our API to server static file from the folder `public`. To enable it, you have to import the `express` module to use the middleware `app.use(express.static("public"))` at the end of the `libs/middlewares.js` file and don't forget to create the `public` empty folder. See how it looks:

```
import bodyParser from "body-parser";
import express from "express";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

To validate if everything is working fine, let's start documenting about the `GET /` endpoint, for this endpoint we'll use the following comments:

- `@api`: informs the type, the address and the title of the endpoint;
- `@apiGroup`: informs the endpoint group name;
- `@apiSuccess`: describes the fields and their data types for a successful's response;
- `@apiSuccessExample`: shows a output's sample of a successful's response.

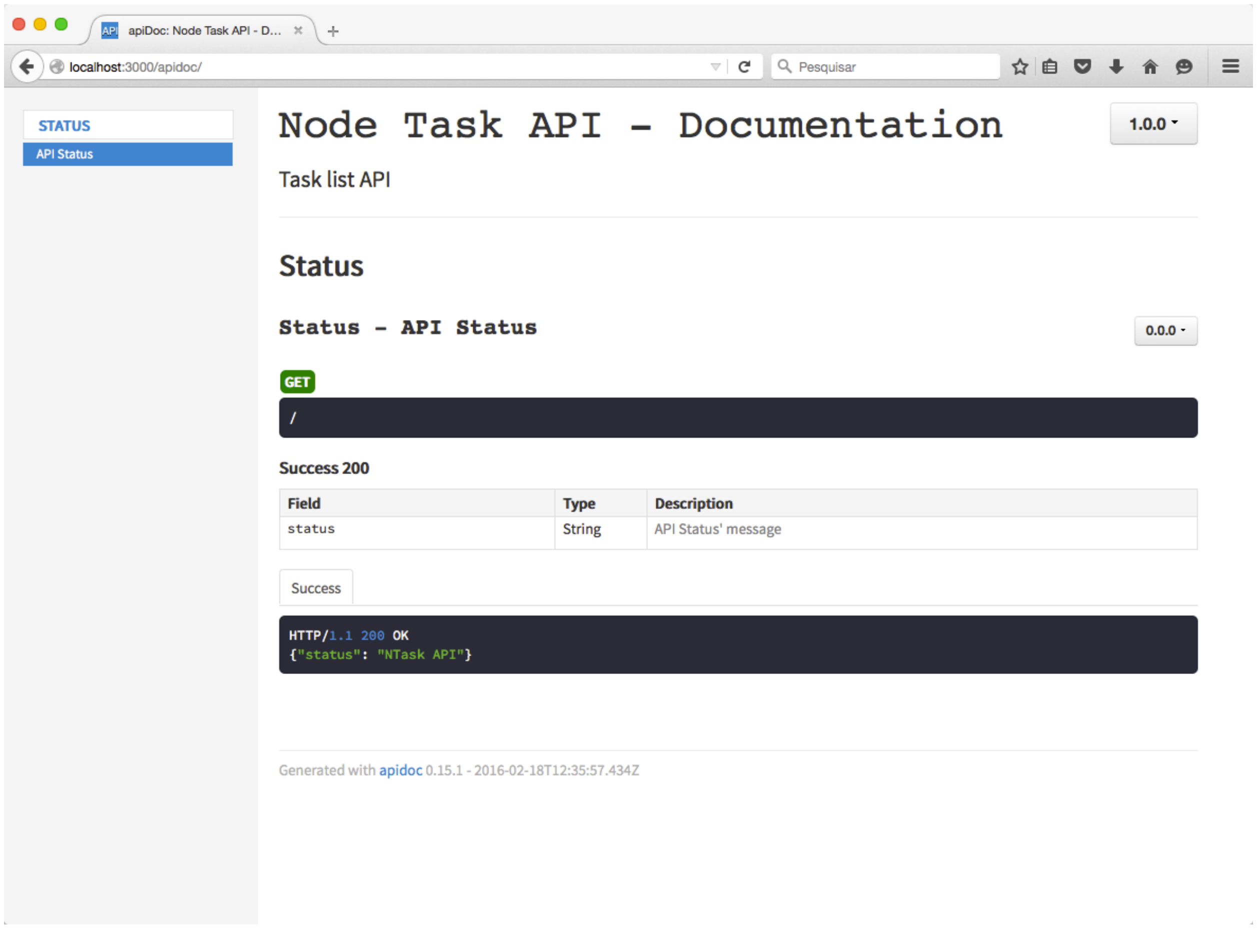
To document this endpoint, edit the file `routes/index.js` writing this code below:

```
module.exports = app => {
  /**
   * @api {get} / API Status
   * @apiGroup Status
   * @apiSuccess {String} status API Status' message
   * @apiSuccessExample {json} Success
   *    HTTP/1.1 200 OK
   *    {"status": "NTask API"}
   */
  app.get("/", (req, res) => {
    res.json({status: "NTask API"});
  });
};
```

To test these changes, restart the server, then open the browser and go to: localhost:3000/apidoc[19].

If no errors occur, you will see a beautiful documentation page about our API.

[19]http://localhost:3000/apidoc


apiDoc: Node Task API - D...

localhost:3000/apidoc/

STATUS

API Status

Node Task API - Documentation

1.0.0

Task list API

Status

Status - API Status

0.0.0

GET

```
/
```

Success 200

Field	Type	Description
status	String	API Status' message

Success

```
HTTP/1.1 200 OK
{"status": "NTask API"}
```

Generated with apidoc 0.15.1 - 2016-02-18T12:35:57.434Z

API documentation page

Documenting token generation

Now, we are going to explore the ApiDoc's features deeper, documenting all API's routes.

To start doing this, let's document the route `POST /token`. This has some extra details to be documented. To do this, we'll use not only the items from the last section but some new ones as well:

- `@apiParam`: describes an input parameter, which may be or not required its submission in a request;
- `@apiParamExample`: shows a sample of a input parameter, in our case, we will display a JSON input format;
- `@apiErrorExample`: shows a sample of some errors which could be generated by the API if something wrong happens.

To understand in practice the usage of these new items, let's edit the `routes/token.js`, following the comments below:

```
import jwt from "jwt-simple";

module.exports = app => {
  const cfg = app.libs.config;
  const Users = app.db.models.Users;

  /**
   * @api {post} /token Authentication Token
   * @apiGroup Credentials
   * @apiParam {String} email User email
   * @apiParam {String} password User password
   * @apiParamExample {json} Input
   *    {
   *      "email": "john@connor.net",
   *      "password": "123456"
   *    }
   * @apiSuccess {String} token Token of authenticated user
   * @apiSuccessExample {json} Success
   *    HTTP/1.1 200 OK
   *    {"token": "xyz.abc.123.hgf"}
   * @apiErrorExample {json} Authentication error
   *    HTTP/1.1 401 Unauthorized
   */
  app.post("/token", (req, res) => {
    // The code here was explained in chapter 7.
  });
};
```

Documenting user's resource

In this section and in the next one, we'll focus to document the main resources of our API. As the majority of the routes of these resources need an authenticated **token** - which is sent in the request's header - we are going to use the following items to describe more about this header:

- `@apiHeader`: describes the name and data type of a header;
- `@apiHeaderExample`: shows a sample of header to be used in the request.

Open the `routes/users.js` and let's doc it! First, we'll doc the `GET /user` endpoint:

```
module.exports = app => {
  const Users = app.db.models.Users;

  app.route("/user")
    .all(app.auth.authenticate())
    /**
     * @api {get} /user Return the authenticated user's data
     * @apiGroup User
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiSuccess {Number} id User id
     * @apiSuccess {String} name User name
     * @apiSuccess {String} email User email
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 200 OK
     *    {
     *      "id": 1,
     *      "name": "John Connor",
     *      "email": "john@connor.net"
     *    }
     * @apiErrorExample {json} Find error
     *    HTTP/1.1 412 Precondition Failed
     */
    .get((req, res) => {
      // GET /user logic...
    })
```

Then, let's doc the route `DELETE /user`:

```
    /**
     * @api {delete} /user Deletes an authenticated user
     * @apiGroup User
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 204 No Content
     * @apiErrorExample {json} Delete error
     *    HTTP/1.1 412 Precondition Failed
     */
    .delete((req, res) => {
```

```
    // DELETE /user logic...
    })
```

To finish this endpoint, we need to doc its last route, the `POST /user`, now we'll use several items to describe the input and output fields of this route:

```
    /**
     * @api {post} /users Register a new user
     * @apiGroup User
     * @apiParam {String} name User name
     * @apiParam {String} email User email
     * @apiParam {String} password User password
     * @apiParamExample {json} Input
     *    {
     *      "name": "John Connor",
     *      "email": "john@connor.net",
     *      "password": "123456"
     *    }
     * @apiSuccess {Number} id User id
     * @apiSuccess {String} name User name
     * @apiSuccess {String} email User email
     * @apiSuccess {String} password User encrypted password
     * @apiSuccess {Date} updated_at Update's date
     * @apiSuccess {Date} created_at Register's date
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 200 OK
     *    {
     *      "id": 1,
     *      "name": "John Connor",
     *      "email": "john@connor.net",
     *      "password": "$2a$10$SK1B1",
     *      "updated_at": "2016-02-10T15:20:11.700Z",
     *      "created_at": "2016-02-10T15:29:11.700Z",
     *    }
     * @apiErrorExample {json} Register error
     *    HTTP/1.1 412 Precondition Failed
     */
    app.post("/users", (req, res) => {
      // POST /users logic...
    });
};
```

Documenting task’s resource

Continuing our API documentation, now we need to finish the task’s routes documentation, let’s edit the `routes/tasks.js` file, initially describing the `GET /tasks` route:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .all(app.auth.authenticate())
    /**
     * @api {get} /tasks List the user's tasks
     * @apiGroup Tasks
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *     {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiSuccess {Object[]} tasks Task's list
     * @apiSuccess {Number} tasks.id Task id
     * @apiSuccess {String} tasks.title Task title
     * @apiSuccess {Boolean} tasks.done Task is done?
     * @apiSuccess {Date} tasks.updated_at Update's date
     * @apiSuccess {Date} tasks.created_at Register's date
     * @apiSuccess {Number} tasks.user_id Id do usuário
     * @apiSuccessExample {json} Success
     *     HTTP/1.1 200 OK
     *     [{
     *       "id": 1,
     *       "title": "Study",
     *       "done": false
     *       "updated_at": "2016-02-10T15:46:51.778Z",
     *       "created_at": "2016-02-10T15:46:51.778Z",
     *       "user_id": 1
     *     }]
     * @apiErrorExample {json} List error
     *     HTTP/1.1 412 Precondition Failed
     */
     .get((req, res) => {
      // GET /tasks logic...
     })
```

Then, let’s doc the route `POST /tasks`:

```
    /**
     * @api {post} /tasks Register a new task
     * @apiGroup Tasks
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {String} title Task title
     * @apiParamExample {json} Input
     *    {"title": "Study"}
     * @apiSuccess {Number} id Task id
     * @apiSuccess {String} title Task title
     * @apiSuccess {Boolean} done=false Task is done?
     * @apiSuccess {Date} updated_at Update's date
     * @apiSuccess {Date} created_at Register's date
     * @apiSuccess {Number} user_id User id
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 200 OK
     *    {
     *      "id": 1,
     *      "title": "Study",
     *      "done": false,
     *      "updated_at": "2016-02-10T15:46:51.778Z",
     *      "created_at": "2016-02-10T15:46:51.778Z",
     *      "user_id": 1
     *    }
     * @apiErrorExample {json} Register error
     *    HTTP/1.1 412 Precondition Failed
     */
     .post((req, res) => {
     // POST /tasks logic...
     })
```

Then, we are going to doc the route `GET /tasks/:id` with the following comments:

```
    /**
     * @api {get} /tasks/:id Get a task
     * @apiGroup Tasks
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {id} id Task id
     * @apiSuccess {Number} id Task id
     * @apiSuccess {String} title Task title
     * @apiSuccess {Boolean} done Task is done?
     * @apiSuccess {Date} updated_at Update's date
     * @apiSuccess {Date} created_at Register's date
     * @apiSuccess {Number} user_id User id
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 200 OK
     *    {
     *      "id": 1,
     *      "title": "Study",
     *      "done": false
     *      "updated_at": "2016-02-10T15:46:51.778Z",
     *      "created_at": "2016-02-10T15:46:51.778Z",
     *      "user_id": 1
     *    }
     * @apiErrorExample {json} Task not found error
     *    HTTP/1.1 404 Not Found
     * @apiErrorExample {json} Find error
     *    HTTP/1.1 412 Precondition Failed
     */
     .get((req, res) => {
      // GET /tasks/:id logic...
     })
```

Now, the `PUT /tasks/:id`:

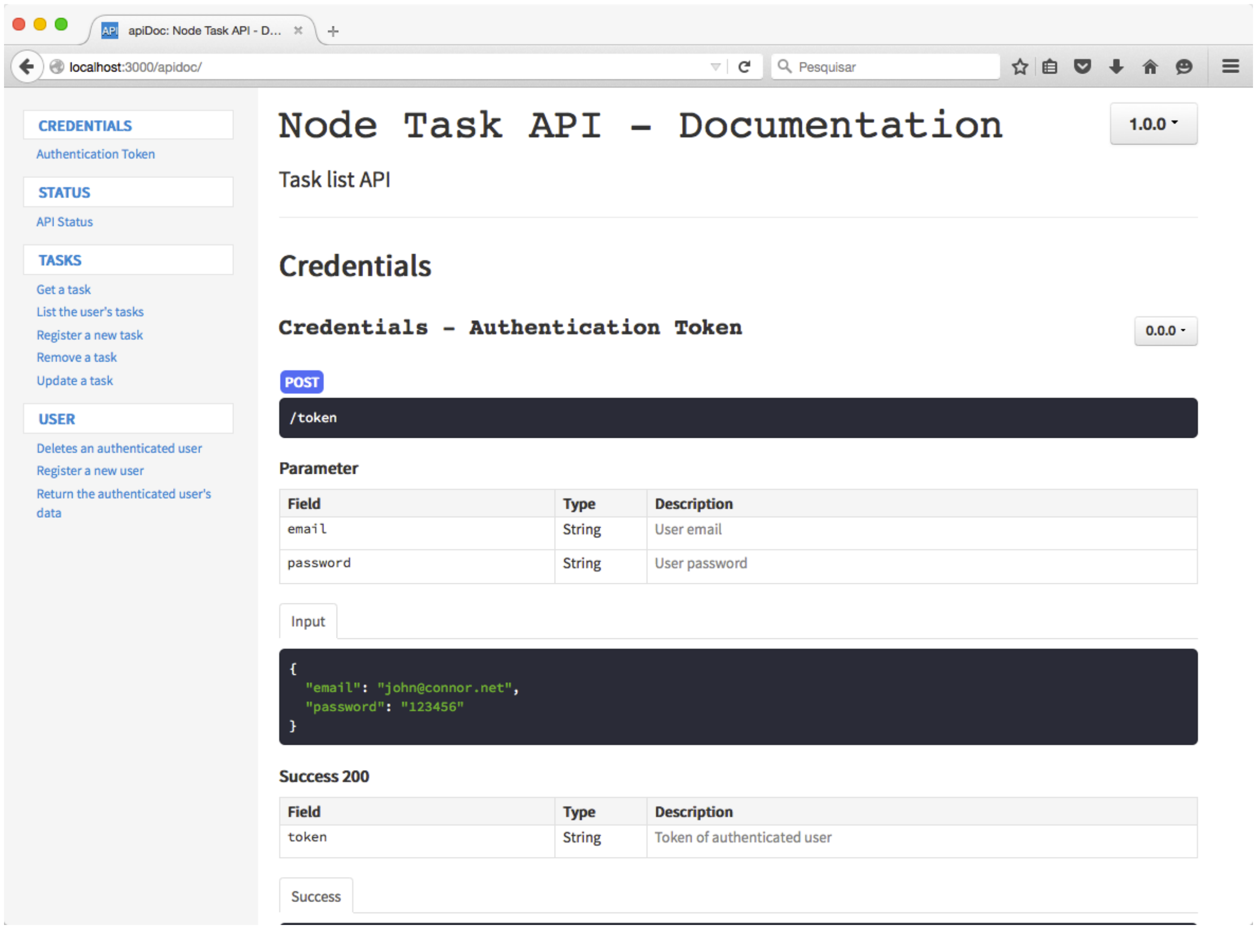
```
    /**
     * @api {put} /tasks/:id Update a task
     * @apiGroup Tasks
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {id} id Task id
     * @apiParam {String} title Task title
     * @apiParam {Boolean} done Task is done?
     * @apiParamExample {json} Input
     *    {
     *      "title": "Work",
     *      "done": true
     *    }
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 204 No Content
     * @apiErrorExample {json} Update error
     *    HTTP/1.1 412 Precondition Failed
     */
     .put((req, res) => {
      // PUT /tasks/:id logic...
     })
```

At last, let's finish this chapter documenting the route `DELETE /tasks/:id`:

```
    /**
     * @api {delete} /tasks/:id Remove a task
     * @apiGroup Tasks
     * @apiHeader {String} Authorization Token of authenticated user
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {id} id Task id
     * @apiSuccessExample {json} Success
     *    HTTP/1.1 204 No Content
     * @apiErrorExample {json} Delete error
     *    HTTP/1.1 412 Precondition Failed
     */
     .delete((req, res) => {
      // DELETE /tasks/:id logic...
     });
};
```

Let's test? Just restart the server and then go to: localhost:3000/apidoc[20].

This time, we have a complete documentation page that describes step-by-step for a new developer how to create a client application to consume data from our API.



Node Task API - Documentation

Task list API

Credentials

Credentials - Authentication Token

```
/token
```

Parameter

Field	Type	Description
email	String	User email
password	String	User password

```
{
  "email": "john@connor.net",
  "password": "123456"
}
```

Success 200

Field	Type	Description
token	String	Token of authenticated user

Now our API is well documented

Conclusion

Congratulations! We have finished another excellent chapter. Now we not only have a functional API as well as a complete documentation to allow other developers to create client-side applications using our API.

Keep reading because, in the next episode, we'll include some frameworks and best practices for our API works fine in a production environment.

[20]http://localhost:3000/apidoc

Preparing the production environment

Introduction to CORS

In case you do not know, CORS *(Cross-origin resource sharing)* is an important HTTP mechanism. It is responsible for allowing or not asynchronous requests from other domains.

CORS, in practice, are only the HTTP's headers that are included on server-side. Those headers can inform which domain can consume the API, which HTTP methods are allowed and, mainly, which endpoints can be shared publicly to applications from other domains consume.

Enabling CORS in the API

As we are developing an API that will serve data for any kind of client-side applications, we need to enable the CORS's middleware for the endpoints become public. Meaning that any client can make requests on our API. To enable it, let's install and use the module `cors`:

```
npm install cors@2.7.1 --save
```

Then, to initiate it, just use the middleware `app.use(cors())` in the `libs/middlewares.js`:

```
import bodyParser from "body-parser";
import express from "express";
import cors from "cors";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(cors());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

When using only the function `cors()` the middleware will release full access of our API. However, it is advisable to control which client domains can have access and which methods they can use, and, mainly, which headers must be required to the clients inform in the request. In our case, let's setup only three attributes: `origin` (allowed domains), `methods` (allowed methods) e `allowedHeaders` (requested headers).

In the `libs/middlewares.js`, let's add these parameters in `app.use(cors())` function:

```
import bodyParser from "body-parser";
import express from "express";
import cors from "cors";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Now we have an API that is only going to allow client apps from the address: localhost:3001[21].

Keep calm, because this will be the domain of our client application that we'll build in the next chapters.

A bit more about CORS

For study purposes about CORS, to understand its headers and, most important, to learn how to customize the rule for your API, I recommend you to read this full documentation: https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS[22]

[21]http://localhost:3001

[22]https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Generating logs

In this section, we are going to setup our application to report and generate logs files about the user's requests. To do this, let's use the module `winston` that is useful in treat several kinds of logs.

In our case, the logs requests will be treated by the module `morgan` which is a middleware for generating request's logs in the server. We also will get logs for any SQL command from database to generate a full log in the API. First, install the modules `winston` and `morgan`.

```
npm install winston@2.1.1 morgan@1.6.1 --save
```

After that, let's write the code responsible to setup and load the `winston` module. We are going to verify if the logs' folder exist, using the native module `fs` *(File System)* to treat files.

Then, will be implemented a simple conditional via function `fs.existsSync("logs")`, to check if there is or not a `logs` folder. If it doesn't exist, it will be created by `fs.mkdirSync("logs")` function.

After this verification, we'll load and export the `module.exports = new winston.Logger` to be able to generate logs files. The logs object will use as `transports` the object `new winston.transports.File`, which is responsible for creating and maintaining several recent logs files. Create the file `libs/logger.js` like this:

```
import fs from "fs";
import winston from "winston";

if (!fs.existsSync("logs")) {
  fs.mkdirSync("logs");
}

module.exports = new winston.Logger({
  transports: [
    new winston.transports.File({
      level: "info",
      filename: "logs/app.log",
      maxsize: 1048576,
      maxFiles: 10,
      colorize: false
    })
  ]
});
```

Now, we are going to use our `libs/logger.js` for two important aspect of our application. First, to generate SQL logs commands modifying the file `libs/config.development.js` to load our module `logger`. We'll use its function `logger.info()` as a callback of the attribute `logging` from Sequelize to capture the SQL command via `logger.info()` function. To do this, just edit the `libs/config.development.js` as below:

```
import logger from "./logger.js";

module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    logging: (sql) => {
      logger.info(`[${new Date()}] ${sql}`);
    },
    define: {
      underscored: true
    }
  },
  jwtSecret: "Nta$K-AP1",
  jwtSession: {session: false}
};
```

To finish, we are going to use the module `logger` to generate the request's logs. To do this, let's use the module `morgan` and include in the top of the middlewares the function `app.use(morgan("common"))` to log all requests.

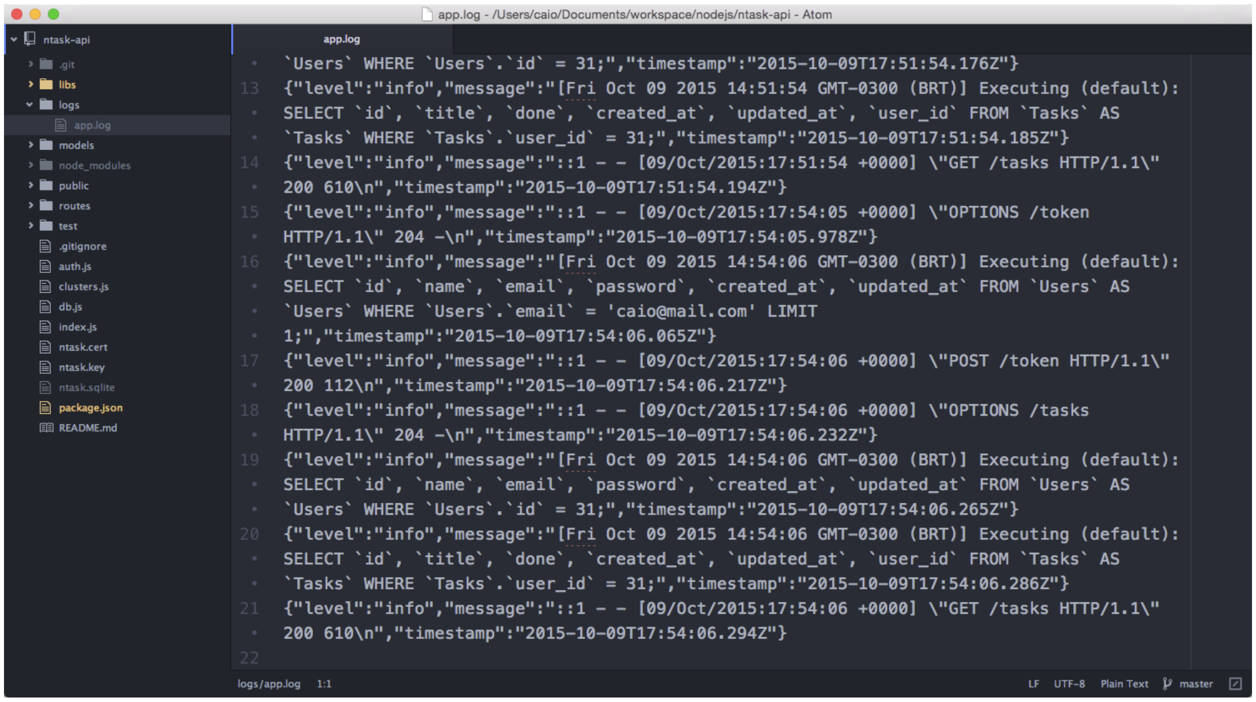
To send these logs to our module `logger`, add the attribute `stream` as a callback function called `write(message)` and, then, send the variable `message` to our log function, the `logger.info(message)`. To understand this implementation better, edit the file `libs/middlewares.js` as the following code below:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

To test the log generation, restart the server and access many times any API address, such as localhost:3000[23].

After making some requests, access the `logs` folder. There will be a file containing the logs of the requests similar to this one:

[23]http://localhost:3000



```
`Users` WHERE `Users`.`id` = 31;","timestamp":"2015-10-09T17:51:54.176Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:51:54 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `title`, `done`, `created_at`, `updated_at`, `user_id` FROM `Tasks` AS
`Tasks` WHERE `Tasks`.`user_id` = 31;","timestamp":"2015-10-09T17:51:54.185Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:51:54 +0000] \"GET /tasks HTTP/1.1\"
200 610\n","timestamp":"2015-10-09T17:51:54.194Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:05 +0000] \"OPTIONS /token
HTTP/1.1\" 204 -\n","timestamp":"2015-10-09T17:54:05.978Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `name`, `email`, `password`, `created_at`, `updated_at` FROM `Users` AS
`Users` WHERE `Users`.`email` = 'caio@mail.com' LIMIT
1;","timestamp":"2015-10-09T17:54:06.065Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"POST /token HTTP/1.1\"
200 112\n","timestamp":"2015-10-09T17:54:06.217Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"OPTIONS /tasks
HTTP/1.1\" 204 -\n","timestamp":"2015-10-09T17:54:06.232Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `name`, `email`, `password`, `created_at`, `updated_at` FROM `Users` AS
`Users` WHERE `Users`.`id` = 31;","timestamp":"2015-10-09T17:54:06.265Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `title`, `done`, `created_at`, `updated_at`, `user_id` FROM `Tasks` AS
`Tasks` WHERE `Tasks`.`user_id` = 31;","timestamp":"2015-10-09T17:54:06.286Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"GET /tasks HTTP/1.1\"
200 610\n","timestamp":"2015-10-09T17:54:06.294Z"}
```

Log of the requests

Configuring parallel processing using cluster module

Unfortunately, Node.js does not work with multi-threads. This is something that in the opinion of some developers is considered a negative aspect and that causes disinterest in learning and in taking it seriously. However, despite being single-thread, it's possible to prepare it to work at least with parallel processing. To do this, you can use the native module called `cluster`.

It basically instantiates new processes of an application working in a distributed way and this module takes care to share the same port network between the actives clusters. The number of processes to be created is you who determines, but a good practice is to instantiate a number of processes based in the amount of server processor cores, or also a relative amount to **core x processors**.

For example, if I have a single processor of eight cores, you can instantiate eight processes, creating a **network of eight clusters**. But, if there is four processors of each cores each, you can create a **network of thirty-two clusters** in action.

To make sure the clusters work in a distributed and organized way, it is necessary that a **parent process** exists (a.k.a **cluster master**). Because it is responsible for balancing the parallel processing among the others clusters, distributing this load to the other processes, called **child process** (a.k.a **cluster slave**). It is very easy to implement this technique on Node.js, since all processing distribution is performed abstractedly to the developer.

Another advantage is that the clusters are independent. That is, in case a cluster goes down, the others will continue working. However, it is necessary to manage the instances and the shutdown of clusters manually to ensure the return of the cluster that went down.

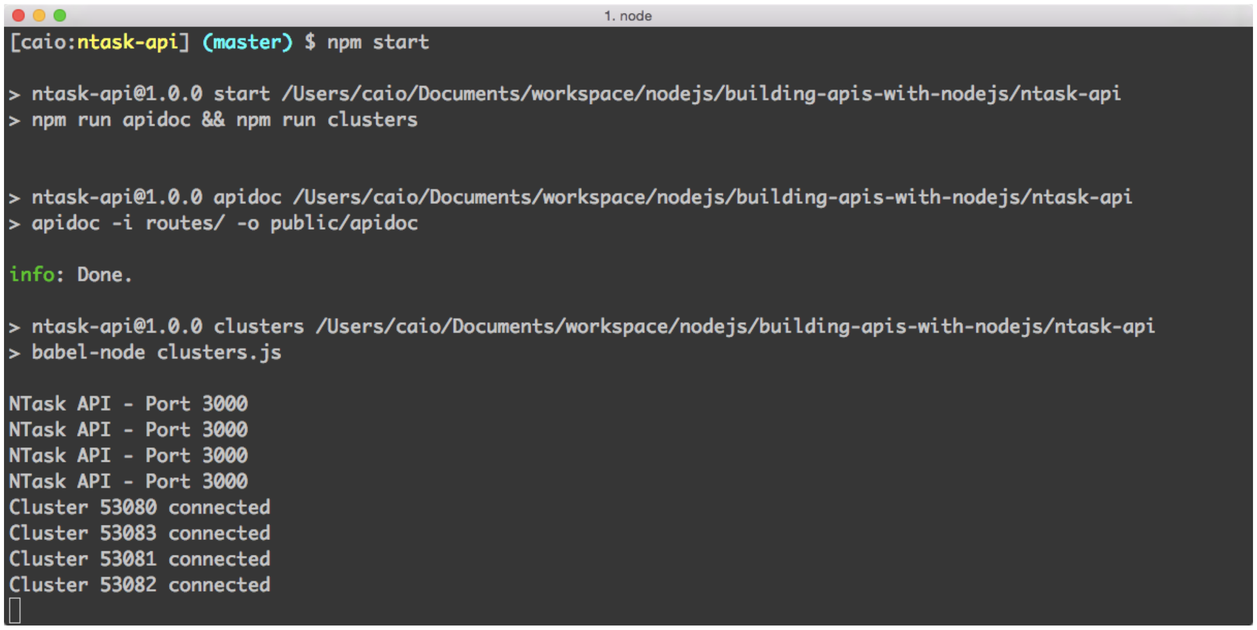
Basing ourselves on these concepts, we are going to apply in practice the implementation of clusters. Create in the root directory the file `clusters.js`. See the code bellow:

```
import cluster from "cluster";
import os from "os";

const CPUS = os.cpus();
if (cluster.isMaster) {
  CPUS.forEach(() => cluster.fork());
  cluster.on("listening", worker => {
    console.log("Cluster %d connected", worker.process.pid);
  });
  cluster.on("disconnect", worker => {
    console.log("Cluster %d disconnected", worker.process.pid);
  });
  cluster.on("exit", worker => {
    console.log("Cluster %d is dead", worker.process.pid);
    cluster.fork();
    // Ensure to starts a new cluster if an old one dies
  });
} else {
  require("./index.js");
}
```

This time, to setup the server, first edit the `package.json` inside the attribute `scripts` and create the command `npm run clusters`, as the code below:

```
"scripts": {
  "start": "npm run apidoc && babel-node index.js",
  "clusters": "babel-node clusters.js",
  "test": "NODE_ENV=test mocha test/**/*.js",
  "apidoc": "apidoc -i routes/ -o public/apidoc"
}
```

Now, if you execute the command `npm run clusters`. This time, the application will run distributed into the clusters and to make sure it worked, you will see the message `"NTask API - Port 3000"` more than once in the terminal.



```
[caio:ntask-api] (master) $ npm start

> ntask-api@1.0.0 start /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-api
> npm run apidoc && npm run clusters


> ntask-api@1.0.0 apidoc /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-api
> apidoc -i routes/ -o public/apidoc

info: Done.

> ntask-api@1.0.0 clusters /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-api
> babel-node clusters.js

NTask API - Port 3000
NTask API - Port 3000
NTask API - Port 3000
NTask API - Port 3000
Cluster 53080 connected
Cluster 53083 connected
Cluster 53081 connected
Cluster 53082 connected
```

Running Node.js in cluster mode

Basically, we have to load the module `cluster` and first, verify it is the **master cluster** via `cluster.isMaster` variable. If the cluster is master, a loop will be iterated based on the total of processing cores **(CPUs)** forking new slave clusters inside the `CPUS.forEach(() => cluster.fork())` function.

When a new process is created (in this case a child process), consequently it does not fit in the conditional `if(cluster.isMaster)`. So, the application server is started via `require("./index.js")` for this child process.

Also, some events created by the **cluster master** are included. In the last code, we only used the main events, listed below:

- `listening`: happens when a cluster is listening a port. In this case, our application is listening the port **3000**;
- `disconnect`: happens when a cluster is disconnected from the cluster's network;
- `exit`: occurs when a cluster is closed in the OS.

Developing clusters

A lot of things can be explored about developing clusters on Node.js. Here, we only applied a little bit which was enough to run parallel processing. But, in case you have to implement a more detailed clusters, I recommend you to read the documentation: nodejs.org/api/cluster.html[24].

To finish and automate the command `npm start` to starts the server in cluster mode, let's update the `package.json` file, following the code below:

[24]https://nodejs.org/api/cluster.html

```
"scripts": {
  "start": "npm run apidoc && npm run clusters",
  "clusters": "babel-node clusters.js",
  "test": "NODE_ENV=test mocha test/**/*.js",
  "apidoc": "apidoc -i routes/ -o public/apidoc"
}
```

Done! Now you can start the clusters network using the command `npm start`.

Compacting requests using GZIP middleware

To make the requests lighter and load faster, let's enable another middleware which is going to be responsible to compact the JSON responses and also the entire API documentation static files to GZIP format – a compatible format to several browsers. We'll do this simple, but important refactoring just using the module `compression`. So let's install it:

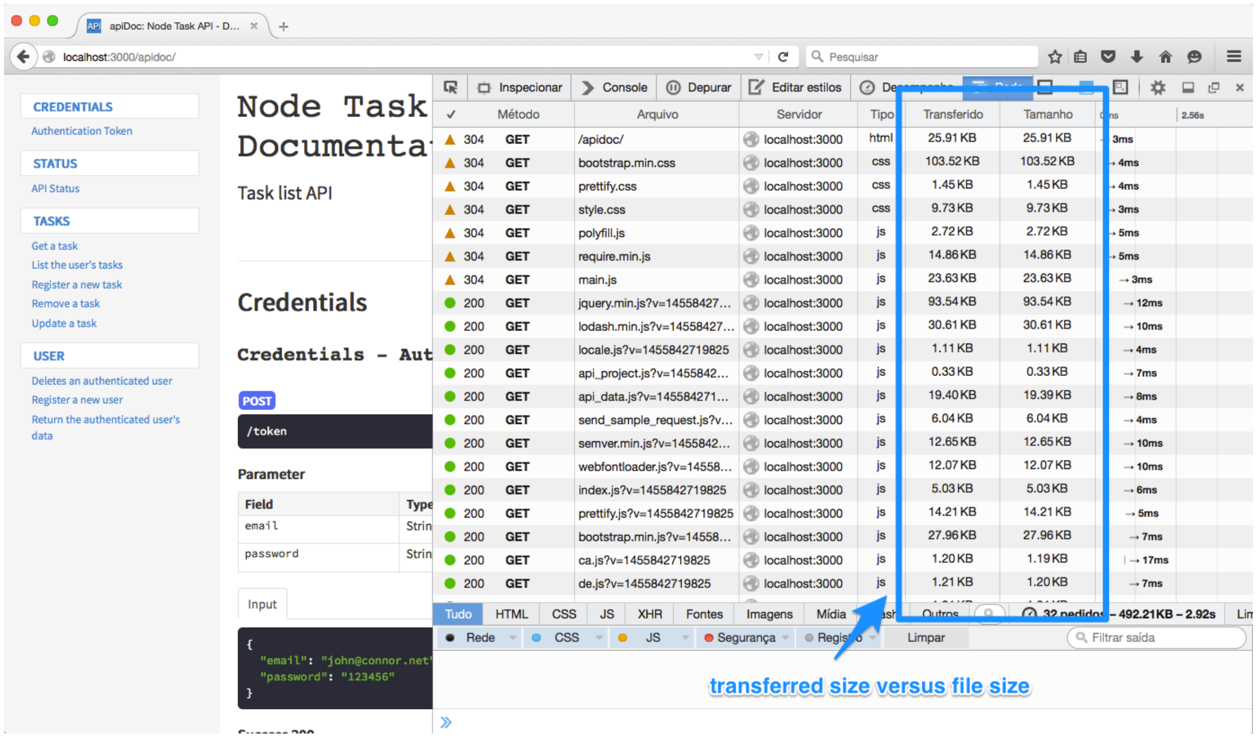
```
npm install compression@1.6.1 --save
```

After installing it, will be necessary to include its middleware into `libs/middlewares.js` file:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import compression from "compression";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(compression());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

To test this implementation, restart the server and then go to the API documentation, after all, there are a lot of static file there, in the address: localhost:3000/apidoc[25]

To see in details, open the browser console (Firefox and Google Chrome have greats client-side consoles) and go to the **Networks** menu. There, you can see the **transferred size versus file size**, it's something similar to this image:



```
{
  "email": "john@connor.net",
  "password": "123456"
}
```

✓	Método	Arquivo	Servidor	Tipo	Transferido	Tamanho	0 ms
304	GET	/apidoc/	localhost:3000	html	25.91 KB	25.91 KB	3ms
304	GET	bootstrap.min.css	localhost:3000	css	103.52 KB	103.52 KB	4ms
304	GET	prettify.css	localhost:3000	css	1.45 KB	1.45 KB	4ms
304	GET	style.css	localhost:3000	css	9.73 KB	9.73 KB	3ms
304	GET	polyfill.js	localhost:3000	js	2.72 KB	2.72 KB	5ms
304	GET	require.min.js	localhost:3000	js	14.86 KB	14.86 KB	5ms
304	GET	main.js	localhost:3000	js	23.63 KB	23.63 KB	3ms
200	GET	jquery.min.js?v=14558427…	localhost:3000	js	93.54 KB	93.54 KB	12ms
200	GET	lodash.min.js?v=14558427…	localhost:3000	js	30.61 KB	30.61 KB	10ms
200	GET	locale.js?v=1455842719825	localhost:3000	js	1.11 KB	1.11 KB	4ms
200	GET	api_project.js?v=1455842…	localhost:3000	js	0.33 KB	0.33 KB	7ms
200	GET	api_data.js?v=145584271…	localhost:3000	js	19.40 KB	19.39 KB	8ms
200	GET	send_sample_request.js?v…	localhost:3000	js	6.04 KB	6.04 KB	4ms
200	GET	semver.min.js?v=1455842…	localhost:3000	js	12.65 KB	12.65 KB	10ms
200	GET	webfontloader.js?v=14558…	localhost:3000	js	12.07 KB	12.07 KB	10ms
200	GET	index.js?v=1455842719825	localhost:3000	js	5.03 KB	5.03 KB	6ms
200	GET	prettify.js?v=1455842719825	localhost:3000	js	14.21 KB	14.21 KB	5ms
200	GET	bootstrap.min.js?v=14558…	localhost:3000	js	27.96 KB	27.96 KB	7ms
200	GET	ca.js?v=1455842719825	localhost:3000	js	1.20 KB	1.19 KB	17ms
200	GET	de.js?v=1455842719825	localhost:3000	js	1.21 KB	1.20 KB	7ms

Applying GZIP in some files

[25] http://localhost:3000/apidoc

Installing SSL support to use HTTPS

Nowadays, it is required to build a safe application that has a safe connection between the server and the client. To do this, many applications buy and use security certificates to ensure a SSL **(Secure Sockets Layer)** connection via the HTTPS protocol.

To implement a HTTPS protocol connection, it is necessary to buy a digital certificate for production's environment usage. But, in our case, we are gonna work with a fake certificate, not valid for production usage, but valid enough for didactic ends. To create a simple certificate, you can go to: www.selfsignedcertificate.com[26]

Informs `ntask` domain and click on **Generate** button. A new screen will be displayed with two extension files: `.key` and `.cert`. Download these files and put them into the root folder of our project.

Now, let's use the native `https` module to allow our server to start using HTTPS protocol and the `fs` module to read the downloaded files: `ntask.key` and `ntask.cert` to be used as credential parameters to start our server in HTTPS mode. To do this, we are going to replace the function `app.listen()` to `https.createServer(credentials, app).listen()` function.. To implement this change, edit the `libs/boot.js`:

```
import https from "https";
import fs from "fs";

module.exports = app => {
  if (process.env.NODE_ENV !== "test") {
    const credentials = {
      key: fs.readFileSync("ntask.key", "utf8"),
      cert: fs.readFileSync("ntask.cert", "utf8")
    }
    app.db.sequelize.sync().done(() => {
      https.createServer(credentials, app)
        .listen(app.get("port"), () => {
          console.log(`NTask API - Port ${app.get("port")}`);
        });
    });
  }
};
```

Congratulations! Now your application is running in a safe protocol, ensuring that the data won't be intercepted. Note that in a real project this kind of implementation requires a valid digital certificate, so don't forget to buy one if you put a serious API in a production's environment.

To test this changes, just restart and go to: https://localhost:3000[27]

[26]http://www.selfsignedcertificate.com
[27]https://localhost:3000

Armoring the API with Helmet

Finishing the development of our API, let's include a very important module, which is a security middleware that handles several kinds of attacks in the HTTP/HTTPS protocols. This module is called `helmet` which is a set of nine internal middlewares, responsible to treat the following HTTP settings:

- Configures the **Content Security Policy**;
- Remove the header `X-Powered-By` that informs the name and the version of a server;
- Configures rules for **HTTP Public Key Pinning**;
- Configures rules for **HTTP Strict Transport Security**;
- Treats the header `X-Download-Options` for Internet Explorer 8+;
- Disable the **client-side caching**;
- Prevents **sniffing** attacks on the client **Mime Type**;
- Prevents **ClickJacking** attacks;
- Protects against XSS **(Cross-Site Scripting)** attacks.

To sum up, even if you do not understand a lot about HTTP security, use `helmet` modules, because in addition to have a simple interface, it will armor your web application against many kinds of attacks. To install it, run the command:

```
npm install helmet@1.1.0 --save
```

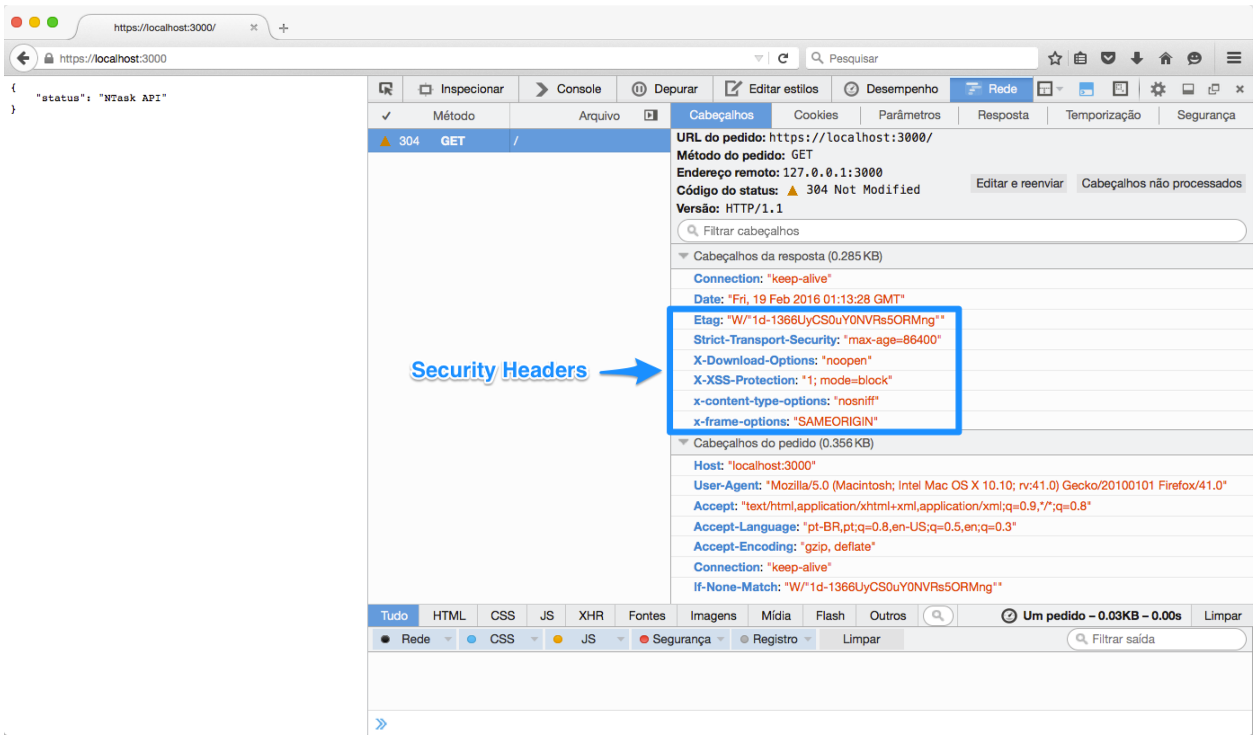
To ensure maximum security on our API, we are going to use all middlewares provided by `helmet` module which can easily be included via function `app.use(helmet())`. So, edit the `libs/middlewares.js` file:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import compression from "compression";
import helmet from "helmet";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(helmet());
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(compression());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Now, restart your application and go to: https://localhost:3000[28]

Open the browser console and, in the **Networks** menu, you can view in details the `GET /` request's data. There, you'll see new items included in the header, something similar to this image:

[28]https://localhost:3000



```
{
  "status": "NTask API"
}
```

Método	Arquivo
304 GET	/

```
URL do pedido: https://localhost:3000/
Método do pedido: GET
Endereço remoto: 127.0.0.1:3000
Código do status: 304 Not Modified
Versão: HTTP/1.1

Cabeçalhos da resposta (0.285 KB)
  Connection: "keep-alive"
  Date: "Fri, 19 Feb 2016 01:13:28 GMT"
  Etag: "W/"1d-1366UyCS0uY0NVRs5ORMng""
  Strict-Transport-Security: "max-age=86400"
  X-Download-Options: "noopen"
  X-XSS-Protection: "1; mode=block"
  x-content-type-options: "nosniff"
  x-frame-options: "SAMEORIGIN"
Cabeçalhos do pedido (0.356 KB)
  Host: "localhost:3000"
  User-Agent: "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:41.0) Gecko/20100101 Firefox/41.0"
  Accept: "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
  Accept-Language: "pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3"
  Accept-Encoding: "gzip, deflate"
  Connection: "keep-alive"
  If-None-Match: "W/"1d-1366UyCS0uY0NVRs5ORMng""

Um pedido – 0.03KB – 0.00s
```

Security headers

Conclusion

Congrats! We have just finished the complete development of our API! You can use this project as a base for your future APIs, because we have developed a documented API which applies the main REST standards, has endpoints well tested, persist data in a SQL database via Sequelize.js module and, mostly important, follows some best practices about performance and security for production's environments.

But, the book is not over yet! In the next chapter, we'll create a web application, which is going to consume our API. It will be a simple SPA **(Single Page Application)** and will be developed using only JavaScript ES6 code too.

Building the client-side app - Part 1

After a long reading about how to build a REST API using Node.js using JavaScript ES6 language, now, from this chapter until the end of this book, we'll create a new project. This time, a front-end project using the best of JavaScript!

This book presented 80% of the content about back-end development, but now, we are gonna focus on the 20% part, which is about front-end development. After all, we have an API, but we need a client-side app, because it's the only way for users interact with our project.

For this reason, in these last chapters, we are going to build a SPA application, using only pure JavaScript. That's right! Only Vanilla JavaScript!

No front-end framework like Angular, Backbone, Ember, React, jQuery or any others will be used. Only the best of Vanilla JavaScript, or should I say Vanilla ES6?

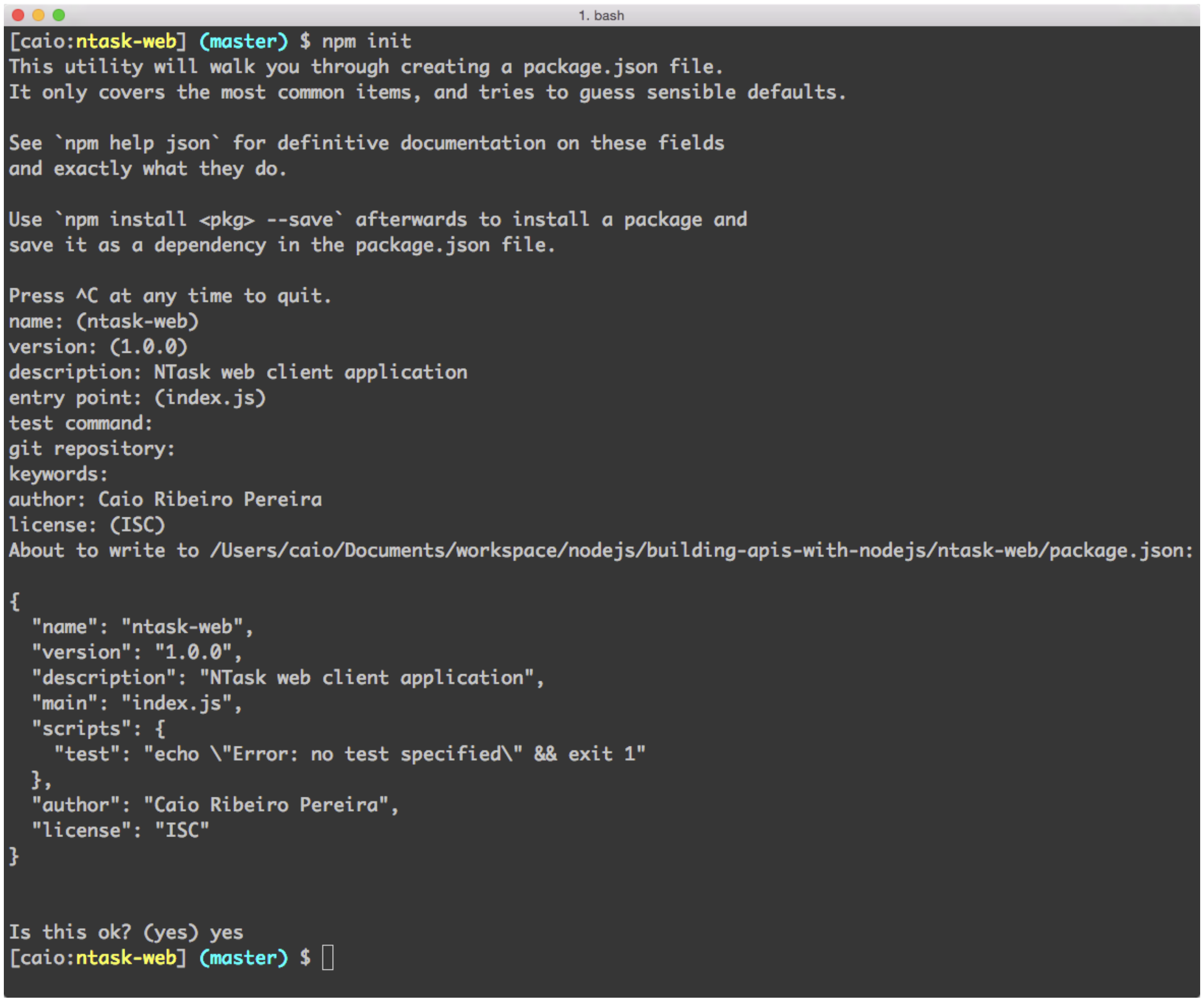
Setting up the app environment

Our client-side app will be built under OOP *(Object-Oriented Programming)* paradigm using ES6 classes and Browserify to be able to use some front-end modules from NPM. Also, we are going to automate some tasks to build our application using only npm alias command, like we used in the API project.

To kick off, let's open the terminal to create a new project. This new project cannot be in the same API directory. This new project will be named `ntask-web`, so, let's go:

```
mkdir ntask-web
cd ntask-web
npm init
```

With the `npm init` command, we are gonna answer these questions to describe this new project:



```
[caio:ntask-web] (master) $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (ntask-web)
version: (1.0.0)
description: NTask web client application
entry point: (index.js)
test command:
git repository:
keywords:
author: Caio Ribeiro Pereira
license: (ISC)
About to write to /Users/caio/Documents/workspace/nodejs/building-apis-with-nodejs/ntask-web/package.json:

{
  "name": "ntask-web",
  "version": "1.0.0",
  "description": "NTask web client application",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Caio Ribeiro Pereira",
  "license": "ISC"
}


Is this ok? (yes) yes
[caio:ntask-web] (master) $
```

NTask Web generating package.json file

After executing the `npm init`, the file `package.json` will be generated. To organize this new project, we must to create some directories in the project's root, these folders are:

- `public`: for static files like css, fonts, javascript and images;
- `public/css`: CSS files only (we are gonna use Ionic CSS);
- `public/fonts`: for font files (we are gonna use Ionic FontIcons);
- `public/js`: For JS files, here we'll put a compiled and minified JS file, generated from the `src` folder;
- `src/components`: for components business logic;
- `src/templates`: for components views;

To create these folders, just run this command:

```
mkdir public/{css,fonts,js}
mkdir src/{components,templates}
```

Below, there is a list with all modules which will be used in this project:

- `http-server`: CLI for start a simple HTTP server for static files only.
- `browserify`: a JavaScript compiler which allows to use some NPM modules, we'll use some modules which have JavaScript Universal codes (codes who works in back-end and front-end layers) and also allows to load using CommonJS pattern, which is the same standard from Node.js.
- `babelify`: a plugin for `browserify` module to enable read and compile ES6 code in the front-end.
- `babel-preset-es2015`: the presets for babel recognizes ES6 code via `babelify`
- `tiny-emitter`: a small module to create and handle events.
- `browser-request`: is a version of `request` module focused for browsers, it is a *cross-browser* (compatible with all major browsers) and abstracts the whole complexity of an AJAX's request.

Basically, we are going to build a web client using these modules. So, let's install them running these commands:

```
npm install http-server@0.9.0 browserify@13.0.0 --save
npm install babelify@7.2.0 babel-preset-es2015@6.5.0 --save
npm install tiny-emitter@1.0.2 browser-request@0.3.3 --save
```

After the installation, let's modify `package.json`, removing the attributes `main`, `script.test` and `license`, and add all the alias commands which will be necessary to automate the build process of this project.

Basically, we'll do this tasks:

- Compile and concatenate all ES6 codes from `src` folder via `npm run build` command;
- Initiate the server in the port `3001` via `npm run server` command;
- Create the command `npm start`, which will run: `npm run build` and `npm run server` commands;

To apply these alterations, edit the `package.json`:

```json
{
  "name": "ntask-web",
  "version": "1.0.0",
  "description": "NTask web client application",
  "scripts": {
    "start": "npm run build && npm run server",
    "server": "http-server public -p 3001",
    "build": "browserify src -t babelify -o public/js/app.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babelify": "^7.2.0",
    "browser-request": "^0.3.3",
    "browserify": "^13.0.0",
    "http-server": "^0.9.0",
    "tiny-emitter": "^1.0.2"
  }
}
```

Now, we need to create again, the `.babelrc` file, to setup the `babel-preset-es2015` module, to do this simple task, create the `.babelrc` in root folder using this code:

```json
{
  "presets": ["es2015"]
}
```

After this setup, let's include some static files that will be responsible for the layout of our project. To not waist time, we are gonna use a ready CSS stylization, from the Ionic framework, a very cool framework with several mobile components ready to build responsive apps.

We won't use the full Ionic framework, because it has a strong dependency of Angular framework. We are only gonna use the CSS file and font icon package. To do this, I recommend you to download the files from the link listed below:

- Ionic CSS: code.ionicframework.com/1.0.0/css/ionic.min.css[29]
- Ionic CSS icons: code.ionicframework.com/ionicons/2.0.0/css/ionicons.min.css[30]
- Ionic Font icons: code.ionicframework.com/1.0.0/fonts/ionicons.eot[31]
- Ionic Font icons: code.ionicframework.com/1.0.0/fonts/ionicons.svg[32]

[29]http://code.ionicframework.com/1.0.0/css/ionic.min.css
[30]http://code.ionicframework.com/ionicons/2.0.0/css/ionicons.min.css
[31]http://code.ionicframework.com/1.0.0/fonts/ionicons.eot
[32]http://code.ionicframework.com/1.0.0/fonts/ionicons.svg

- Ionic Font icons: code.ionicframework.com/1.0.0/fonts/ionicons.ttf[33]
- Ionic Font icons: code.ionicframework.com/1.0.0/fonts/ionicons.woff[34]

And put their CSS files into `public/css` folder and put the fonts files into `public/fonts`.

To start coding, let's create the homepage and the JavaScript code responsible to load the main page, let's do it, just to test if everything will run fine. The main HTML is going to be responsible to load the Ionic CSS stylization, the main JavaScript files and also few HTML tags enough to build the layout structure. To understand this implementation, create the file `public/index.html` using the code below:

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>NTask - Task manager</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link rel="stylesheet" href="css/ionic.min.css">
    <link rel="stylesheet" href="css/ionicons.min.css">
    <script src="js/app.js" async defer></script>
  </head>
  <body>
    <header class="bar bar-header bar-calm">
      <h1 class="title">NTask</h1>
    </header>
    <div class="scroll-content ionic-scroll">
      <div class="content overflow-scroll has-header">
        <main></main>
        <footer></footer>
      </div>
    </div>
  </body>
</html>
```

Note: there are two empty tags: `<main></main>` and `<footer></footer>`. All the page interaction will manipulate these tags dynamically using JavaScript codes from the `src/components` files that we will write soon.

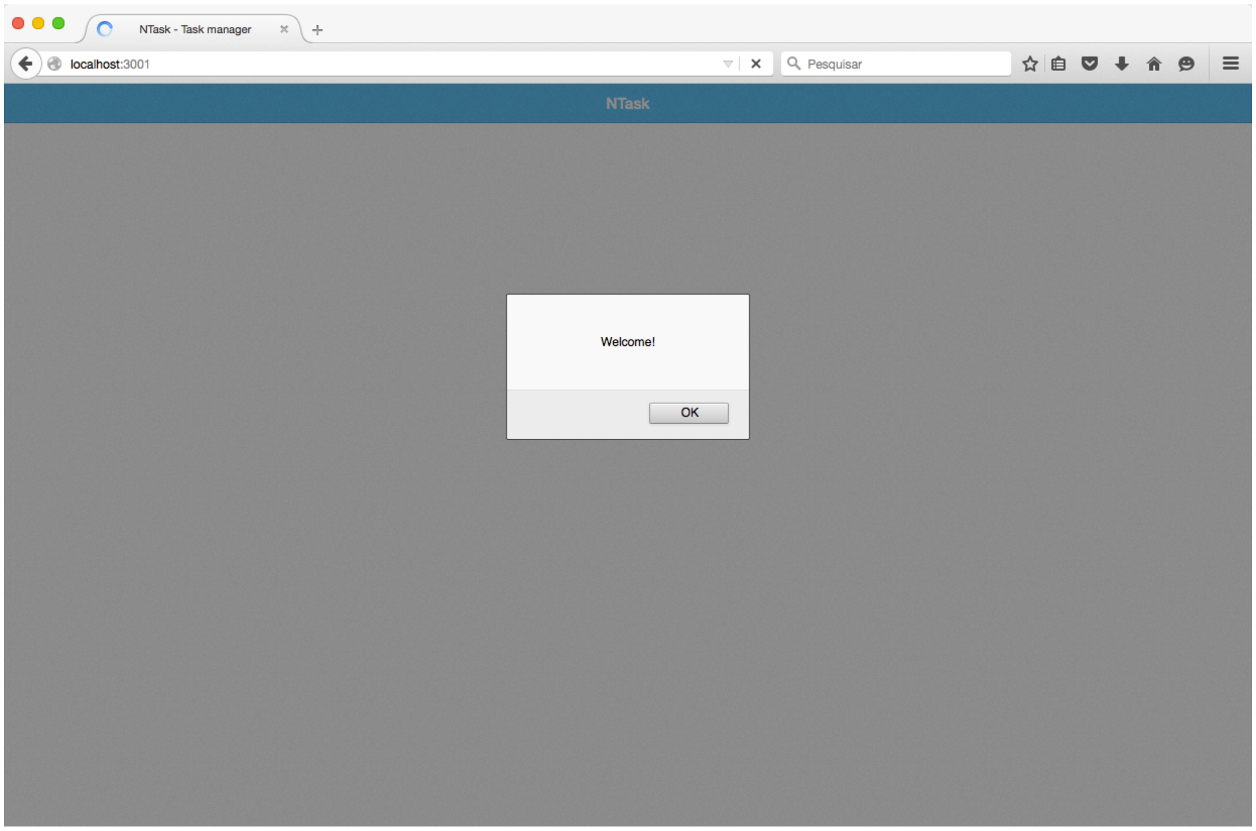
To finish this section, create the `src/index.js` file, for while, this will display a simple `Welcome!` message in the browser when the page is loaded. This will be modified soon, after all, we are going to create it, just to test if our project will work fine.

[33] http://code.ionicframework.com/1.0.0/fonts/ionicons.ttf
[34] http://code.ionicframework.com/1.0.0/fonts/ionicons.woff

```
window.onload = () => {
  alert("Welcome!");
};
```

Now we already have a simple, but functional environment to run our NTask client-side app. To execute it, run the command `npm start` and go to: localhost:3001[35]

If everything works fine, you'll see this result:



First NTask screen

Creating signin and signup views

In this section, we are gonna create all the templates to be used in the application. The templates are basically HTML pieces of strings, manipulated via JavaScript, and are largely used on SPA projects. After all, one of the SPA concepts is to load once all static files to only transfer data frequently between client and server.

All screens transition (template's transition) become a task for client-side apps, causing the server only to work with data and the client to pick up these data to put them into an appropriate screens for users interacts in the application.

[35] http://localhost:3001

Our application is a simple task manager, which has a REST API with endpoints to create, update, delete and list tasks, and register, find and delete an user. All templates will be based on these API's features.

To start, let's build the template that is going to be the **sign in** and **sign up** screens. Thanks to the **Template String** feature provided by ES6, it has become possible to create strings with data concatenation in an elegant way using the syntax:

```
console.log(`Olá ${nome}!`);
```

With this, it's no longer be necessary to use any template engine framework, because we can easily create the templates using a simple function to return a HTML string concatenated with some data.

To understand this implementation better, let's write the sign in homepage templates using the function `render()` to return a HTML string. Create the file `src/templates/signin.js`:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Email</span>
        <input type="text" data-email>
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Password</span>
        <input type="password" data-password>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-home"></i> Login
      </button>
    </div>
  </form>
  <div class="padding">
    <button class="button button-block" data-signup>
      <i class="ion-person-add"></i> Sign up
    </button>
  </div>`;
};
```

Now, to complete the flow, we are also gonna create the sign up screen template. Create the file `src/templates/signup.js`, like this code below:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Name</span>
        <input type="text" data-name>
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Email</span>
        <input type="text" data-email>
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Password</span>
        <input type="password" data-password>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-thumbsup"></i> Register
      </button>
    </div>
  </form>`;
};
```

Done! We already have two important screens of the application. Now, we have to create the interaction code for these screens which they will be responsible to render these templates and, especially, interact with the users and communicate with the API.

Writing signin and signup components

The codes for template's interaction are going to be put into `src/components` folder, but before creating them, let's explore two new JavaScript ES6 features: **classes** and **inheritance**. To write our front-end code more semantic and well organized, we'll create a parent class, which is going to have only two important attributes as inheritance for all components which extend it: `this.URL` (contains the API URL address) and `this.request` (contains the `browser-request` module loaded).

Another detail of this parent class is that it will inherit all functionalities from the `tiny-emitter` module (via `class NTask extends TinyEmitter` declaration), and these functionalities will be passed for their child classes as well, allowing our components to be able to listen and trigger events among the others components. To understand this class better, create the file `src/ntask.js` following the code below:

```
import TinyEmitter from "tiny-emitter";
import request from "browser-request";

class NTask extends TinyEmitter {
  constructor() {
    super();
    this.request = request;
    this.URL = "https://localhost:3000";
  }
}

module.exports = NTask;
```

Now that we have the parent class `NTask`, will be possible to create the component's classes, not only are going to have specific functionalities but also attributes and generic functions inherited by the parent class. That is the components, instead of double or triple the codes; they will reuse the codes.

Let's create our first component for the sign in screen. All component's classes of our application will receive from the constructor the `body` object. This `body` is basically, the DOM object from the tag `<main>` or `<footer>`. All components will inherit from the parent class `NTask`, so, the execution of the function `super()` is required on child class's constructor. We'll use to organize our components the methods: `render()` (to render a template) and `addEventListener()` (to listen and treat any DOM components from the templates).

In this case, we are gonna have two encapsulated events in the methods: `formSubmit()` (responsible to make an user authentication on the API) and `signupClick()` (responsible to render the sign up screen in the browser).

Each component must emit an event via the function `this.emit("event-name")`, because we will create an observer class to listen and delegate tasks among the application's components. A good example of task that is going to be largely used is the template's transitions, which occurs when an user clicks on a template's button, and, the click event is triggered to the observer class listen and delegate a task for a new template be rendered in the browser.

To better understand these rules, create the file `src/components/signin.js` following this large code below:

```
import NTask from "../ntask.js";
import Template from "../templates/signin.js";

class Signin extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-email]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
    this.signupClick();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const email = e.target.querySelector("[data-email]");
      const password = e.target.querySelector("[data-password]");
      const opts = {
        method: "POST",
        url: `${this.URL}/token`,
        json: true,
        body: {
          email: email.value,
          password: password.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 401) {
          this.emit("error", err);
        } else {
          this.emit("signin", data.token);
        }
      });
    });
  }
  signupClick() {
    const signup = this.body.querySelector("[data-signup]");
    signup.addEventListener("click", (e) => {
      e.preventDefault();
      this.emit("signup");
    });
  }
}

module.exports = Signin;
```

Let's also create the `Signup` class. This will follow the same pattern from the `Signin` class. To see how it must be, create the `src/components/signup.js` like this:

```
import NTask from "../ntask.js";
import Template from "../templates/signup.js";

class Signup extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-name]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const name = e.target.querySelector("[data-name]");
      const email = e.target.querySelector("[data-email]");
      const password = e.target.querySelector("[data-password]");
      const opts = {
        method: "POST",
        url: `${this.URL}/users`,
        json: true,
        body: {
          name: name.value,
          email: email.value,
          password: password.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("error", err);
        } else {
          this.emit("signup", data);
        }
      });
    });
  }
}

module.exports = Signup;
```

To finish this initial flow, we still have to create the observer class and then, load it inside `src/index.js` file. This will be the responsible code for starting all the interaction among the application's components. The observer class will be called `App`. His constructor will perform the instance of all components, and will have two main methods: the `init()` (responsible to start the component's interactions) and `addEventListener()` (responsible to listen and treat the component's events). To understand this implementation, let's create the `src/app.js` file, following this code below:

```
import Signin from "./components/signin.js";
import Signup from "./components/signup.js";

class App {
  constructor(body) {
    this.signin = new Signin(body);
    this.signup = new Signup(body);
  }
  init() {
    this.signin.render();
    this.addEventListener();
  }
  addEventListener() {
    this.signinEvents();
    this.signupEvents();
  }
  signinEvents() {
    this.signin.on("error", () => alert("Authentication error"));
    this.signin.on("signin", (token) => {
      localStorage.setItem("token", `JWT ${token}`);
      alert("You are logged in!");
    });
    this.signin.on("signup", () => this.signup.render());
  }
  signupEvents() {
    this.signup.on("error", () => alert("Register error"));
    this.signup.on("signup", (user) => {
      alert(`${user.name} you were registered!`);
      this.signin.render();
    });
  }
}

module.exports = App;
```

Basically, it was created the component's events for successful authentication, authentication error, access the sign up screen, successful sign up and sign up error. The `init()` method starts the homepage, which is the sign in screen, and then, executes the `addEventListener()` method, to listen all the components events that are encapsulated inside the methods `signinEvents()` and `signupEvents()`.

To finish this chapter, edit the `src/index.js` file, to be able to load and initiate the `App` class inside the `window.onload()` event and then, start all the interactive flow of the application:

```
import App from "./app.js"

window.onload = () => {
  const main = document.querySelector("main");
  new App(main).init();
};
```

Let's test it? If you followed step-by-step these implementations, you'll have a basic sign in and sign up flow. To run this application, first, open two terminal: one for API server and the other for the client-side app.

In both terminals, you must run the command `npm start`. Now these applications will be available in the addresses:

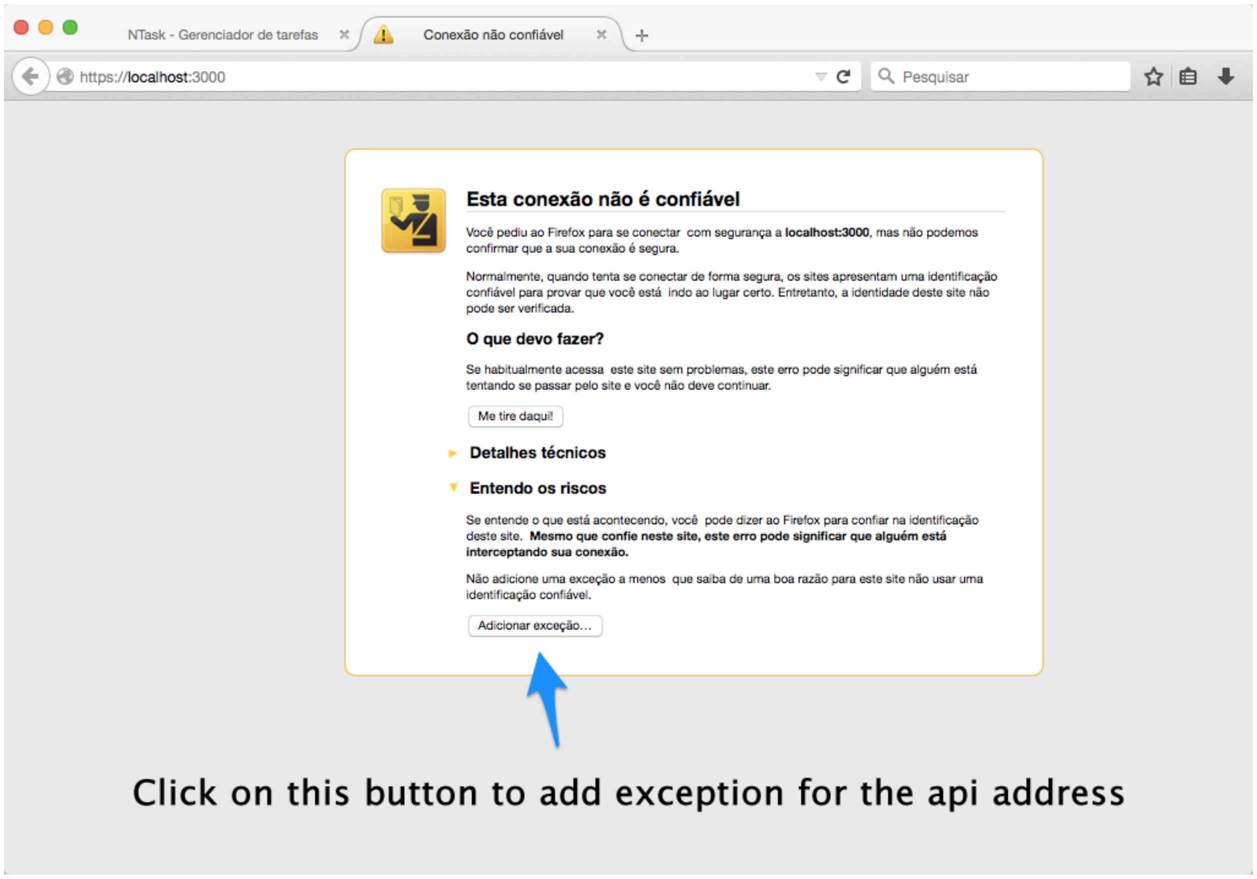
- **NTask API:** https://localhost:3000[36]

[36]https://localhost:3000

- **Ntask Web**: http://localhost:3001[37]

As we are using a not valid digital certificate for production environment, it is quite probable that your browser will block the first access to the API. In case it happens, just open in the browser: https://localhost:3000[38]

Then, and add an exception in your browser to enable the API access. See this image below to understand how to add an exception on Mozilla Firefox browser, for example:



Esta conexão não é confiável

Você pediu ao Firefox para se conectar com segurança a **localhost:3000**, mas não podemos confirmar que a sua conexão é segura.

Normalmente, quando tenta se conectar de forma segura, os sites apresentam uma identificação confiável para provar que você está indo ao lugar certo. Entretanto, a identidade deste site não pode ser verificada.

O que devo fazer?

Se habitualmente acessa este site sem problemas, este erro pode significar que alguém está tentando se passar pelo site e você não deve continuar.

Me tire daqui!

Detalhes técnicos

Entendo os riscos

Se entende o que está acontecendo, você pode dizer ao Firefox para confiar na identificação deste site. **Mesmo que confie neste site, este erro pode significar que alguém está interceptando sua conexão.**

Não adicione uma exceção a menos que saiba de uma boa razão para este site não usar uma identificação confiável.

Adicionar exceção...

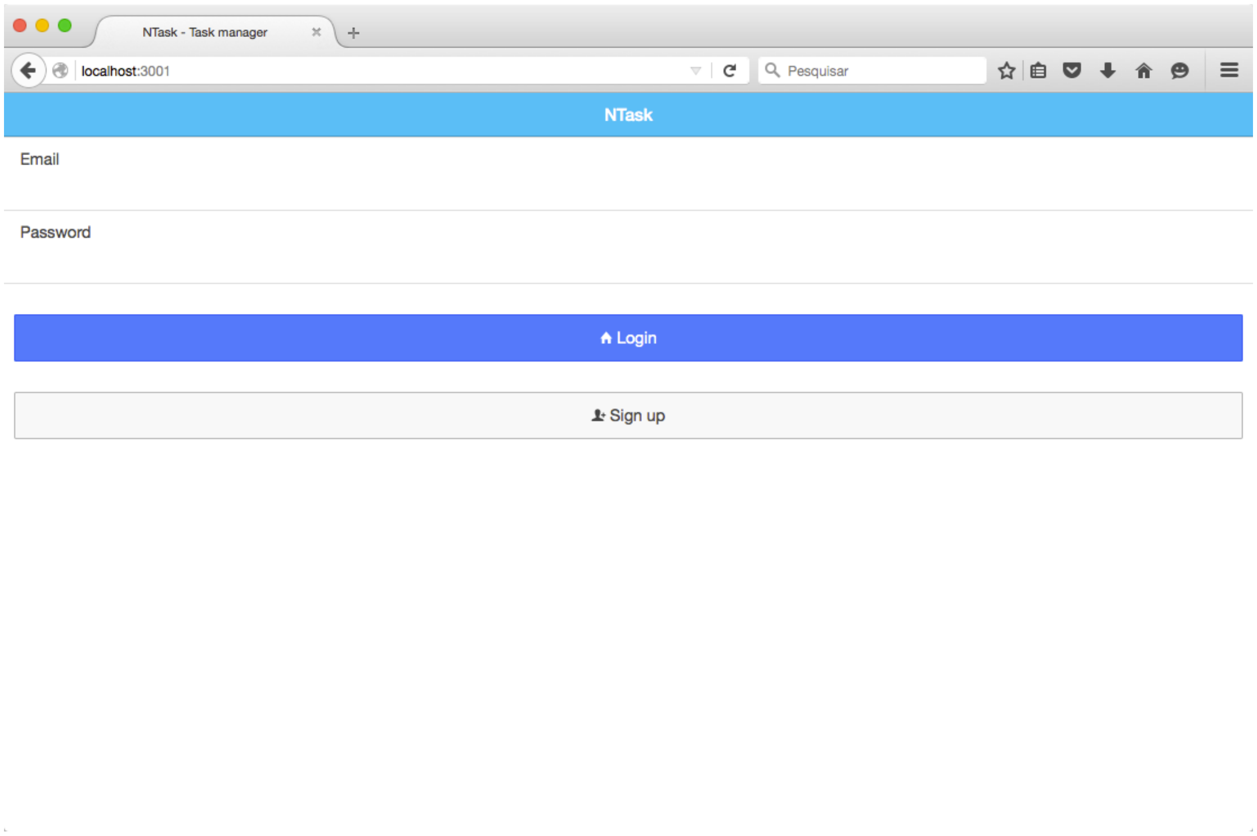
Click on this button to add exception for the api address

Adding and exception to API address

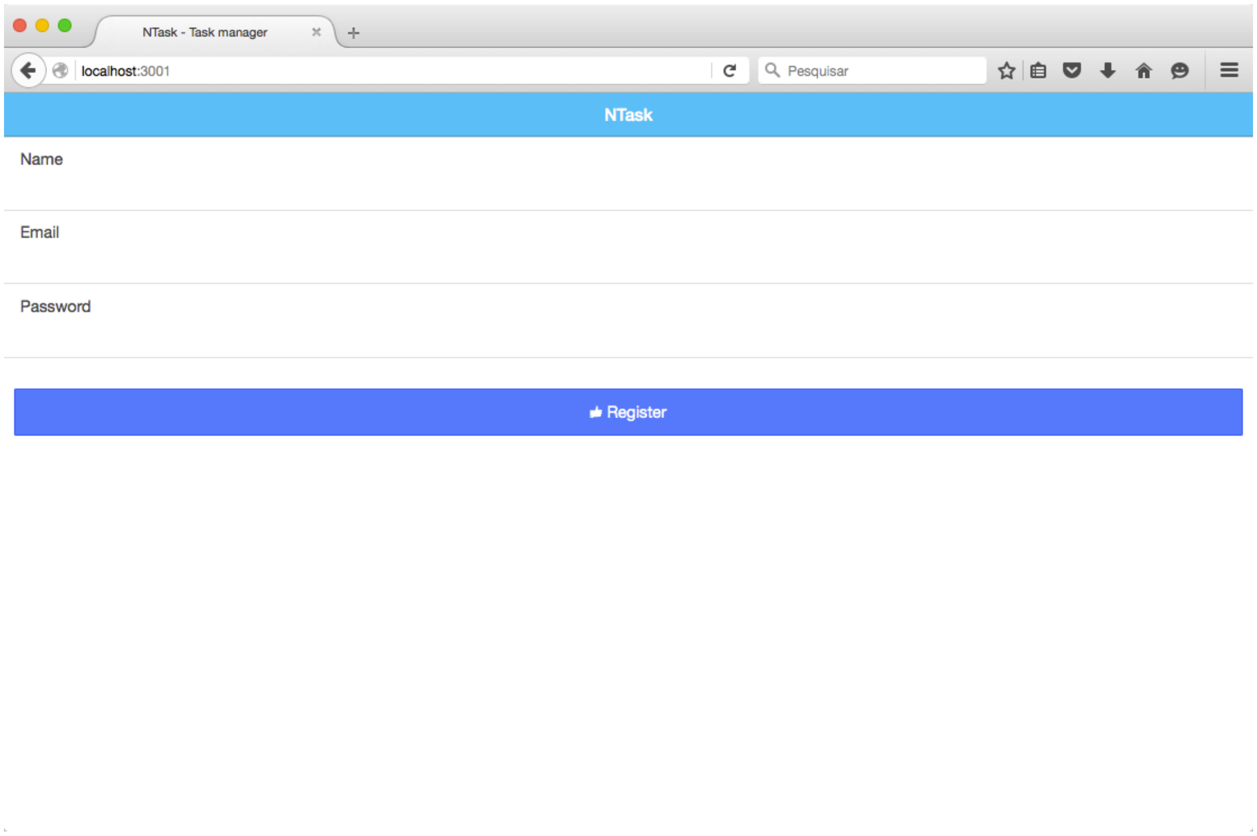
Now that the API has full access, go to the NTask web address: http://localhost:3001[39]

If everything is right, you'll be able to access these screens:

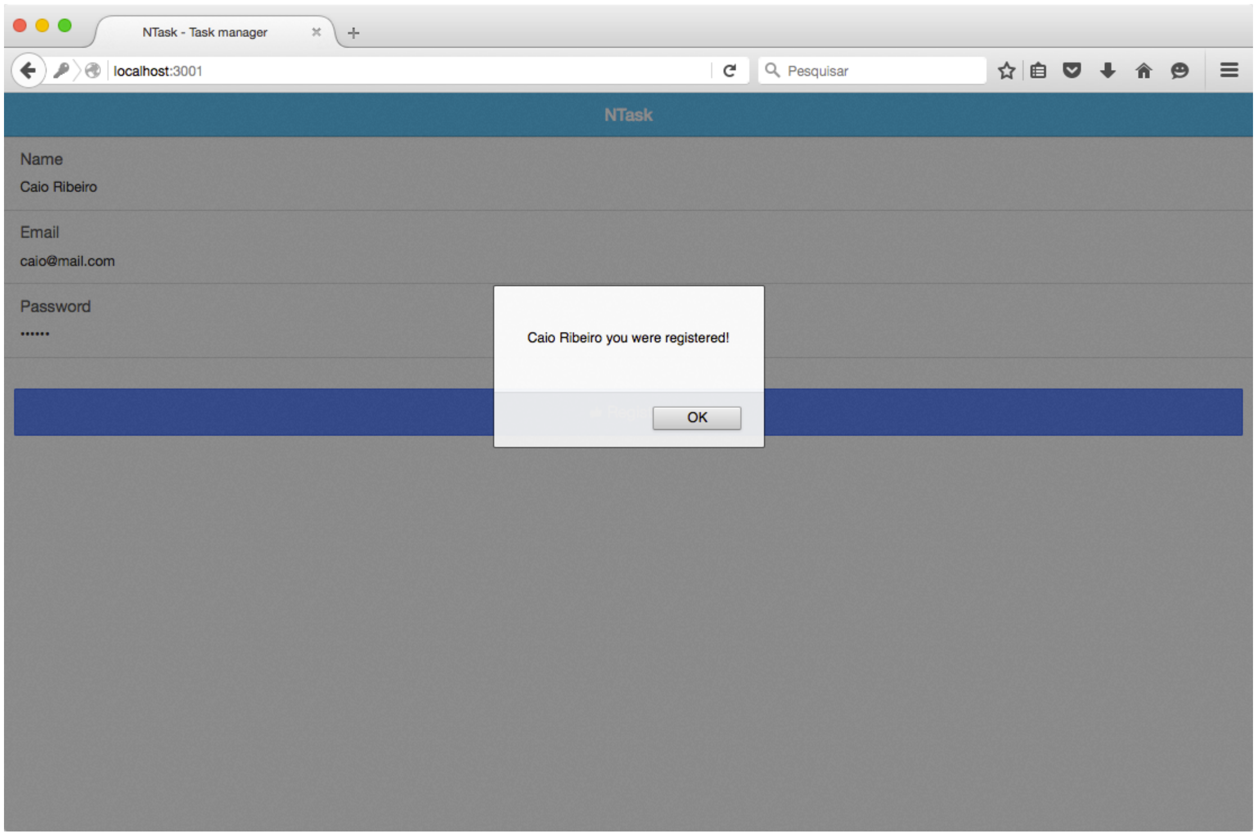
[37] http://localhost:3001
[38] https://localhost:3000
[39] http://localhost:3001



Sign in screen



User sign up screen



Name
Caio Ribeiro

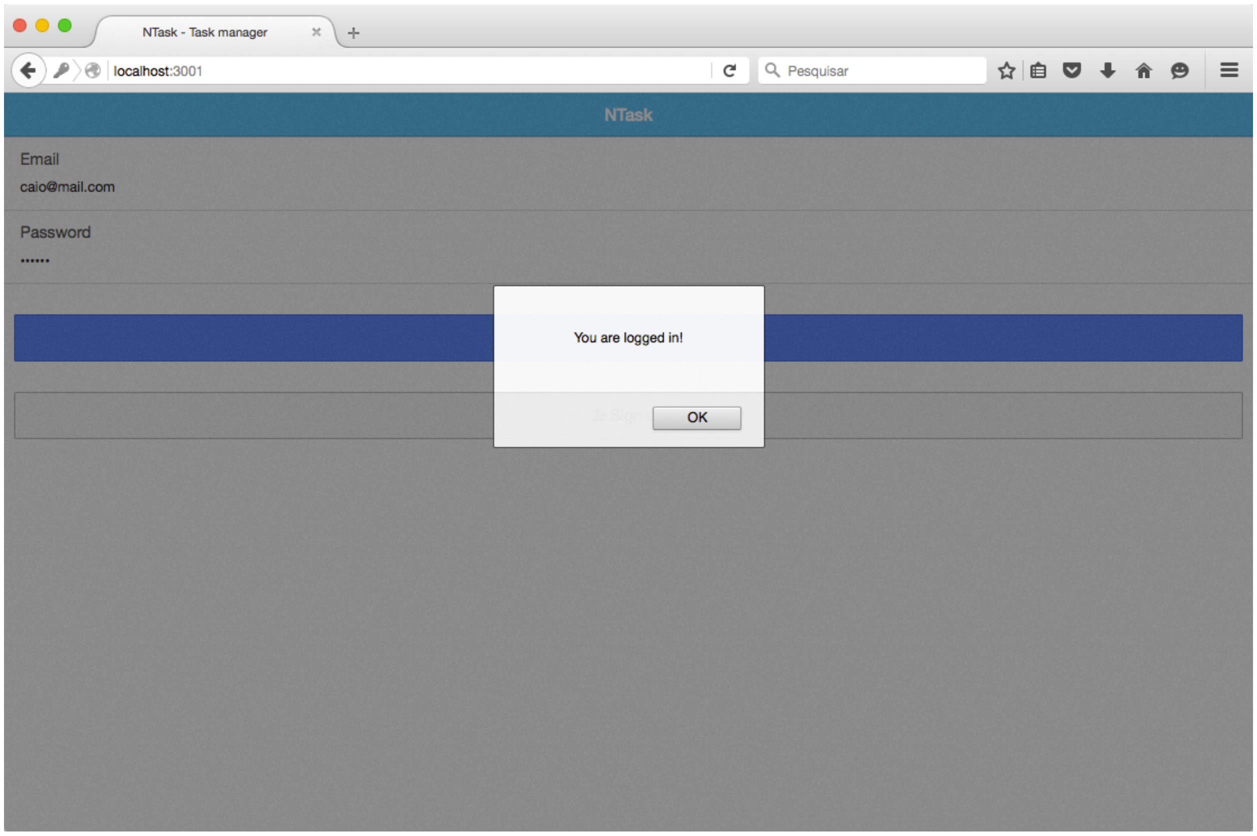
Email
caio@mail.com

Password
••••••

Caio Ribeiro you were registered!

OK

Registering a new user



Logging in with new account

Conclusion

Our new project is taking shape, and we are closer to build a functional system for real users, all of this integrating the client application with the API that was built in the last chapters.

In this chapter, we created only the environment and some screens, enough to structure the application. Keep reading, because in the next chapter we'll go deeper until finish this project. See ya!

Building the client-side app - Part 2

Continuing with the client application construction, up to now we have an application with the main layout that connects with the API and allows to authenticate a user to access the system. In this chapter, we are going to build the main features for users be able to manage their tasks.

Views and components for task's CRUD

The tasks template construction will be quite complex, but in the end, will be great! This template must list all user's tasks, so this function will receive as an argument a tasks list and using the function `tasks.map()` will generate a template's array of the tasks.

In the end of this new array generation, the function `.join("")` will be executed and will concatenate all items returning a single template string of tasks. To make this manipulation and generation of the tasks templates easier, this logic will be encapsulated via `renderTasks(tasks)` function. Otherwise, it displays a message about empty task list.

To understand better this implementation, create the file `src/templates/tasks.js` as below:

```
const renderTasks = tasks => {
  return tasks.map(task => {
    let done = task.done ? "ios-checkmark" : "ios-circle-outline";
    return `<li class="item item-icon-left item-button-right">
      <i class="icon ion-${done}" data-done
        data-task-done="${task.done ? 'done' : ''}"
        data-task-id="${task.id}"></i>
      ${task.title}
      <button data-remove data-task-id="${task.id}"
        class="button button-assertive">
        <i class="ion-trash-a"></i>
      </button>
    </li>`;
  }).join("");
};
exports.render = tasks => {
  if (tasks && tasks.length) {
    return `<ul class="list">${renderTasks(tasks)}</ul>`;
  }
  return `<h4 class="text-center">The task list is empty</h4>`;
};
```

Using the attributes data-task-done, data-task-id, data-done and data-remove, we are gonna work on how to create components for tasks manipulation. These attributes will be used to trigger some events to allow deleting a task (via method taskRemoveClick()) and/or setting which task will be done (via method taskDoneCheckbox()). These business rules will be written on src/components/tasks.js file:

```
import NTask from "../ntask.js";
import Template from "../templates/tasks.js";

class Tasks extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderTaskList();
  }
  addEventListener() {
    this.taskDoneCheckbox();
    this.taskRemoveClick();
  }
  renderTaskList() {
    const opts = {
      method: "GET",
      url: `${this.URL}/tasks`,
      json: true,
      headers: {
        authorization: localStorage.getItem("token")
      }
    };
    this.request(opts, (err, resp, data) => {
      if (err) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  taskDoneCheckbox() {
    const dones = this.body.querySelectorAll("[data-done]");
    for(let i = 0, max = dones.length; i < max; i++) {
      dones[i].addEventListener("click", (e) => {
        e.preventDefault();
        const id = e.target.getAttribute("data-task-id");
        const done = e.target.getAttribute("data-task-done");
        const opts = {
          method: "PUT",
          url: `${this.URL}/tasks/${id}`,
          headers: {
            authorization: localStorage.getItem("token"),
            "Content-Type": "application/json"
          },
          body: JSON.stringify({done: !done})
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("update-error", err);
          } else {
            this.emit("update");
          }
        });
      });
    }
  }
  taskRemoveClick() {
    const removes = this.body.querySelectorAll("[data-remove]");
    for(let i = 0, max = removes.length; i < max; i++) {
      removes[i].addEventListener("click", (e) => {
        e.preventDefault();
        if (confirm("Do you really wanna delete this task?")) {
          const id = e.target.getAttribute("data-task-id");
          const opts = {
            method: "DELETE",
            url: `${this.URL}/tasks/${id}`,
            headers: {
              authorization: localStorage.getItem("token")
            }
          };
          this.request(opts, (err, resp, data) => {
            if (err || resp.status === 412) {
              this.emit("remove-error", err);
            } else {
              this.emit("remove");
            }
        });
      }
    });
  }
 }
}

module.exports = Tasks;
```

Now that we have the component responsible for listing, updating and deleting tasks, let's implement the template and component responsible for adding a new task. This will be easier, because it will be a template with a simple form to register new tasks, and in the end, it redirects to the task list. To do it, create the file `src/templates/taskForm.js`:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Task</span>
        <input type="text" data-task>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-compose"></i> Add
      </button>
    </div>
  </form>`;
};
```

Then, create its respective component which will have only the form submission event from the encapsulated function `formSubmit()`. Create the file `src/components/taskForm.js`:

```
import NTask from "../ntask.js";
import Template from "../templates/taskForm.js";

class TaskForm extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-task]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const task = e.target.querySelector("[data-task]");
      const opts = {
        method: "POST",
        url: `${this.URL}/tasks`,
        json: true,
        headers: {
          authorization: localStorage.getItem("token")
        },
        body: {
          title: task.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("error");
        } else {
          this.emit("submit");
        }
      });
    });
  }
}
```

```

module.exports = TaskForm;
```

Views and components for logged users

To finish the screens creation of our application, we are gonna build the last screen, which will display the logged user's data and a button to be able the user to cancel his account. This screen will have a component very easy to implement as well, because this will only treat the event of the account's cancellation button. Create the `src/templates/user.js`:

```
exports.render = user => {
  return `<div class="list">
    <label class="item item-input item-stacked-label">
      <span class="input-label">Name</span>
      <small class="dark">${user.name}</small>
    </label>
    <label class="item item-input item-stacked-label">
      <span class="input-label">Email</span>
      <small class="dark">${user.email}</small>
    </label>
  </div>
  <div class="padding">
    <button data-remove-account
      class="button button-assertive button-block">
      <i class="ion-trash-a"></i> Cancel account
    </button>
  </div>`;
};
```

Now that we have the user's screen template, let's create its respective component in the `src/components/user.js` file, following the code below:

```
import NTask from "../ntask.js";
import Template from "../templates/user.js";

class User extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderUserData();
  }
  addEventListener() {
    this.userCancelClick();
  }
  renderUserData() {
    const opts = {
      method: "GET",
      url: `${this.URL}/user`,
      json: true,
      headers: {
        authorization: localStorage.getItem("token")
      }
    };
    this.request(opts, (err, resp, data) => {
      if (err || resp.status === 412) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  userCancelClick() {
    const button = this.body.querySelector("[data-remove-account]");
    button.addEventListener("click", (e) => {
      e.preventDefault();
      if (confirm("This will cancel your account, are you sure?")) {
        const opts = {
          method: "DELETE",
          url: `${this.URL}/user`,
          headers: {
            authorization: localStorage.getItem("token")
          }
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("remove-error", err);
          } else {
            this.emit("remove-account");
          }
        });
      }
    });
  }
}

module.exports = User;
```

Creating the main menu

To make this application more elegant and interactive, we are going to also create in its footer the main menu to help users interact with the tasks list and users settings. To create this screen, first we need to create its template, which will have only three buttons: **tasks**, **add task** and **logout**. Create the file `src/templates/footer.js`:

```
exports.render = path => {
  let isTasks = path === "tasks" ? "active" : "";
  let isTaskForm = path === "taskForm" ? "active" : "";
  let isUser = path === "user" ? "active" : "";
  return `
    <div class="tabs-striped tabs-color-calm">
      <div class="tabs">
        <a data-path="tasks" class="tab-item ${isTasks}">
          <i class="icon ion-home"></i>
        </a>
        <a data-path="taskForm" class="tab-item ${isTaskForm}">
          <i class="icon ion-compose"></i>
        </a>
        <a data-path="user" class="tab-item ${isUser}">
          <i class="icon ion-person"></i>
        </a>
        <a data-logout class="tab-item">
          <i class="icon ion-android-exit"></i>
        </a>
      </div>
    </div>`;
};
```

Then, create its corresponding component file: `src/components/menu.js`:

```
import NTask from "../ntask.js";
import Template from "../templates/footer.js";

class Menu extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render(path) {
    this.body.innerHTML = Template.render(path);
    this.addEventListener();
  }
  clear() {
    this.body.innerHTML = "";
  }
  addEventListener() {
    this.pathsClick();
    this.logoutClick();
  }
  pathsClick() {
    const links = this.body.querySelectorAll("[data-path]");
    for(let i = 0, max = links.length; i < max; i++) {
      links[i].addEventListener("click", (e) => {
        e.preventDefault();
        const link = e.target.parentElement;
        const path = link.getAttribute("data-path");
        this.emit("click", path);
      });
    }
  }
  logoutClick() {
    const link = this.body.querySelector("[data-logout]");
    link.addEventListener("click", (e) => {
      e.preventDefault();
      this.emit("logout");
    })
  }
}

module.exports = Menu;
```

Treating all screen’s events

Our project has all the necessary components to build a task list application, now to finish our project we need to assemble all pieces of the puzzle! To start, let’s modify the `src/index.js` so it can manipulate not only the `<main>` tag but also the `<footer>` tag, because this new tag will be used to handle events of the footer menu.

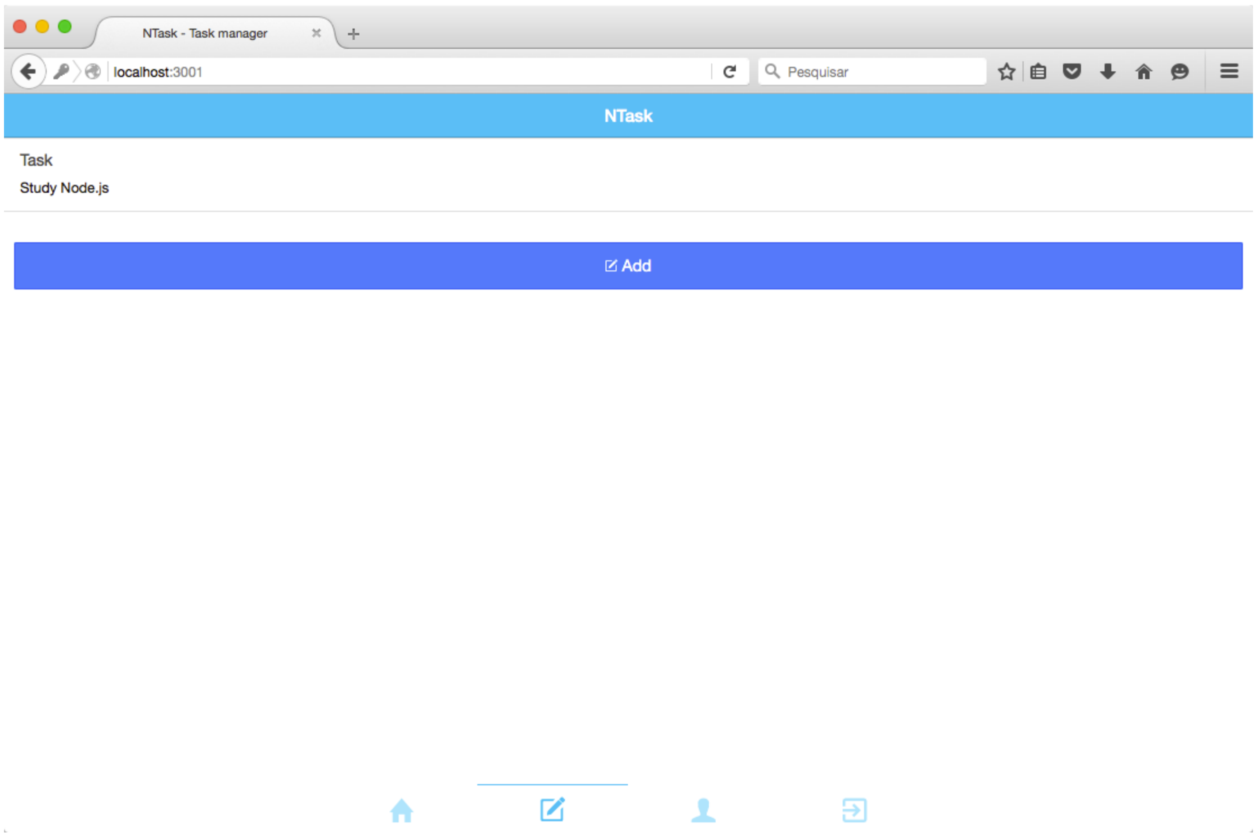
Edit the `src/index.js` applying this simple modification:

```
import App from "./app.js";

window.onload = () => {
  const main = document.querySelector("main");
  const footer = document.querySelector("footer");
  new App(main, footer).init();
};
```

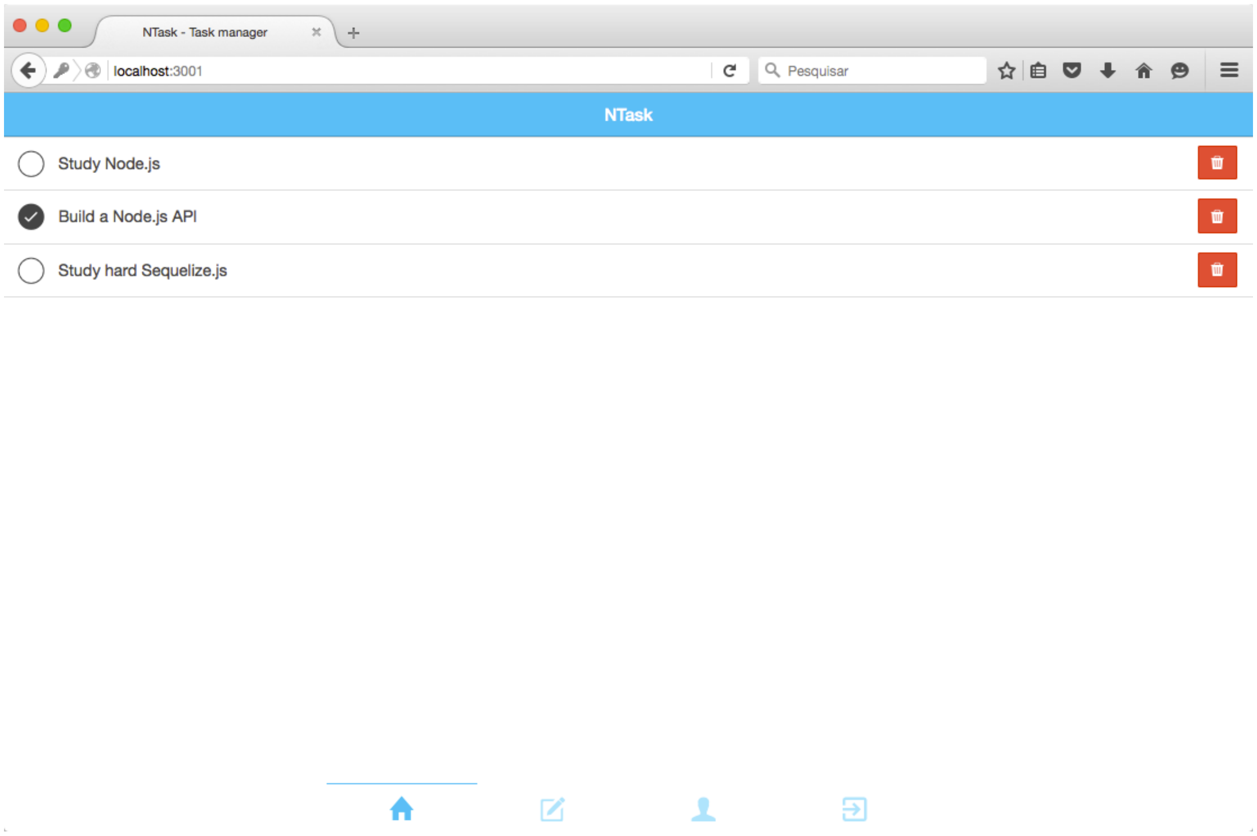
Now, to finish our project, we have to update the object `App` so it can be responsible for loading all the components that were created and, treat the events of each component. Doing this change we will ensure the correct flow for all screen’s transition, the menu transition and the data traffic between the `ntask-api` and `ntask-web` projects. To do this, edit the `src/app.js`, coding this huge script:

```
import Tasks from "./components/tasks.js";
import TaskForm from "./components/taskForm.js";
import User from "./components/user.js";
import Signin from "./components/signin.js";
import Signup from "./components/signup.js";
import Menu from "./components/menu.js";

class App {
  constructor(body, footer) {
    this.signin = new Signin(body);
    this.signup = new Signup(body);
    this.tasks = new Tasks(body);
    this.taskForm = new TaskForm(body);
    this.user = new User(body);
    this.menu = new Menu(footer);
  }
  init() {
    this.signin.render();
    this.addEventListener();
  }
  addEventListener() {
    this.signinEvents();
    this.signupEvents();
    this.tasksEvents();
    this.taskFormEvents();
    this.userEvents();
    this.menuEvents();
  }
  signinEvents() {
    this.signin.on("error", () => alert("Authentication error"));
    this.signin.on("signin", (token) => {
      localStorage.setItem("token", `JWT ${token}`);
      this.menu.render("tasks");
      this.tasks.render();
    });
    this.signin.on("signup", () => this.signup.render());
  }
  signupEvents(){
    this.signup.on("error", () => alert("Register error"));
    this.signup.on("signup", (user) => {
      alert(`${user.name} you were registered!`);
      this.signin.render();
    });
  }
  tasksEvents() {
    this.tasks.on("error", () => alert("Task list error"));
    this.tasks.on("remove-error", () => alert("Task delete error"));
    this.tasks.on("update-error", () => alert("Task update error"));
    this.tasks.on("remove", () => this.tasks.render());
    this.tasks.on("update", () => this.tasks.render());
  }
  taskFormEvents() {
    this.taskForm.on("error", () => alert("Task register error"));
    this.taskForm.on("submit", () => {
      this.menu.render("tasks");
      this.tasks.render();
    });
  }
  userEvents() {
    this.user.on("error", () => alert("User load error"));
    this.user.on("remove-error", () => alert("Cancel account error"));
    this.user.on("remove-account", () => {
      alert("So sad! You are leaving us :(");
      localStorage.clear();
      this.menu.clear();
      this.signin.render();
    });
  }
  menuEvents() {
    this.menu.on("click", (path) => {
      this.menu.render(path);
      this[path].render();
    });
    this.menu.on("logout", () => {
      localStorage.clear();
      this.menu.clear();
      this.signin.render();
    })
  }
}

module.exports = App;
```

Phew! It's done! We built our simple, but useful client application to interact with our current API. Let's test it? You just need to restart the client application and use it normally. Below, you ca see some new screens to access, take a look:


NTask - Task manager
localhost:3001
Pesquisar
NTask
Task
Study Node.js
Add

Adding task

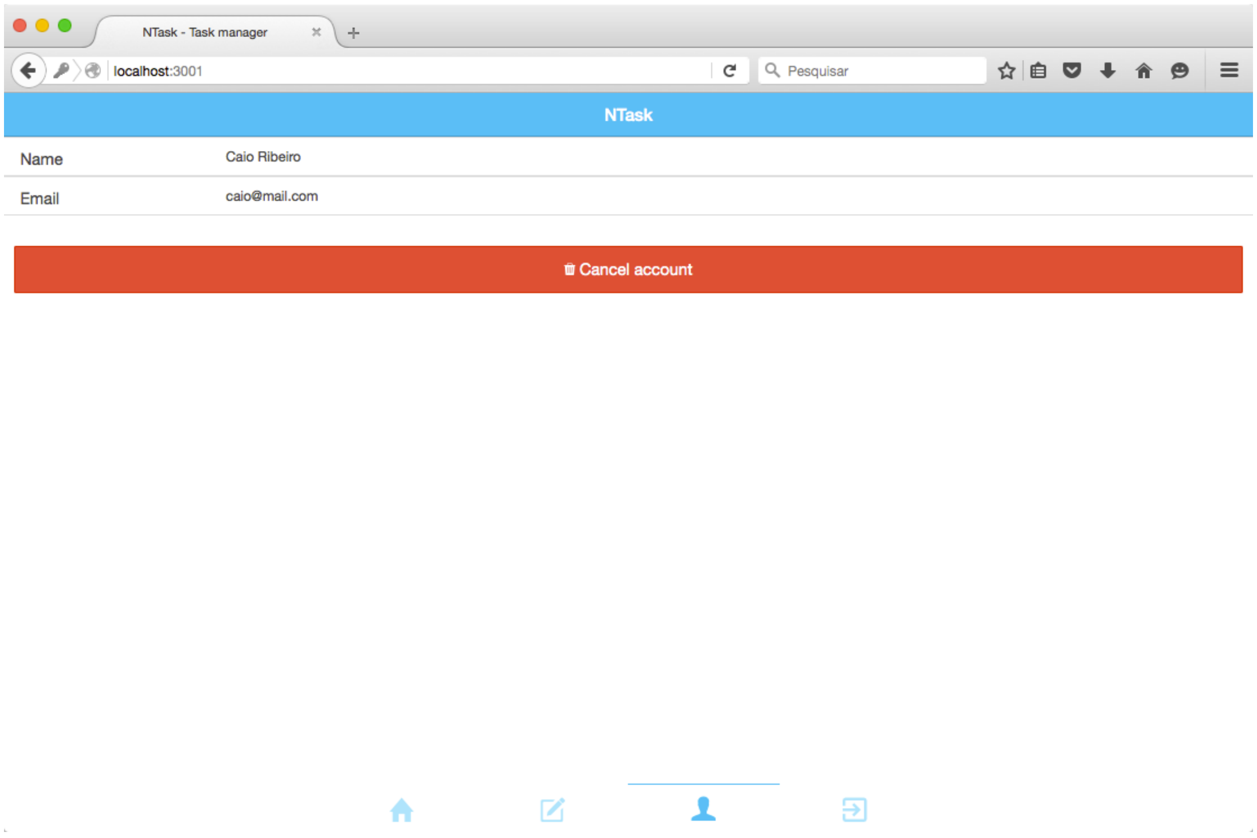


Study Node.js

Build a Node.js API

Study hard Sequelize.js

Listing and checking some tasks



Name	Caio Ribeiro
Email	caio@mail.com

User settings screen

Final conclusion

Congratulations! If you reached this far with your application running perfectly, so you have finished this book successfully. I hope you've learned a lot about Node.js platform reading this book, and especially on how to build a simple, but useful REST API, because that's the essence of the ebook. I believe I have passed the necessary knowledge for you, faithful reader.

Remember that all sources are available into my personal GitHub. Just access this link: github.com/caio-ribeiro-pereira/building-apis-with-nodejs[40].

Thank you very much for reading this book!

[40]https://github.com/caio-ribeiro-pereira/building-apis-with-nodejs