Building APIs with Node.js

Caio Ribeiro Pereira

Apress®

Building APIs with Node.js

Caio Ribeiro Pereira

Apress®

Building APIs with Node.js

Caio Ribeiro Pereira
São Vicente - SP, São Paulo, Brazil

I dedicate this book to my family who always supports me and has motivated me since the beginning of my life.

Contents at a Glance

Contents

About the Author

Caio Ribeiro Pereira is a software engineer from Brazil who love works with Node.js, JavaScript, Meteor, Ruby On Rails, DevOps, and front-end stuffs. He has a bacharelor's degree in Information Systems, author of the blog `https://udgwebdev.com`, creator of the DevFreeBooks (`https://devfreebooks.github.io`) and DevFreeCasts (`https://devfreecasts.github.io`), and is actively engaged in many local meet-ups including NodeBr, DevInSantos, Meteor Brazil and JavaScript Brazil.

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, Web/ and mobile development, and cloud and IT architecture. His true IT passions are security and Android.

He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. He holds a master's of science degree in Computing Science from the University of Salerno, Italy.

He has worked as a project manager, software engineer, research engineer, chief security architect, information security manager, PCI/SCADA auditor, and senior lead IT security/cloud/SCADA architect for many years.

His technical skills include: security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, Web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, and more.

He currently works as Chief Information Security Officer for Cargotec Oyj.

He worked as visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He holds four international patents (PKI, SIP, SAML, and Proxy areas).

Massimo has reviewed more than 40 IT books for different publishing companies and he is the coauthor of *Pro Android Games* (Apress, 2015).

Acknowledgments

Special thanks go to Mrs. Charlotte Bento de Carvalho, my cousin Cláudio Souza, my friends Leandro Alvares da Costa and Bruno Alvares da Costa, Paulo Silveira, and Adriano Almeida. They have had an important role in my life, influencing me to be who I am today, and consequently, to write this book.

Thanks to all the readers from Underground WebDev (`https://udgwebdev.com`); after all, the essence of this book is based on many posts from this blog.

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

Introduction

We live in a time 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 application program interfaces (APIs).

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 interface to the final user. These client-side applications are typically mobile apps, desktop applications, or web apps.

Since 2010 Node.js has increasingly proven to be an excellent platform to solve many problems, especially for building REST APIs. The single-thread architecture that performs nonblocking I/O running on top of JavaScript, which is a ubiquitous language in all current browsers, showed good performance in the processing of many kinds of applications. Some large companies, such as LinkedIn and PayPal, have significantly reduced expenses with servers migrating some of their projects to Node.js.

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

Who Is This Book For?

This book is intended for developers who have at least a basic knowledge of JavaScript and, especially, those who understand object-oriented programming (OOP), a little bit of client-server architecture, and those with an understanding of the main characteristics of REST APIs.

Mastering these concepts, or at minimum a basic knowledge of them, is essential to fully understand this book.

All the code in this book is written using the latest JavaScript implementation, the ECMAScript 2015 (also called ECMAScript 6, or ES6).

How Should I Use This Book?

Throughout the book, many concepts and codes are going to be presented for you to learn through practice all the theoretical parts 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 are both integrated to work as a single project.

It is highly recommended that you follow the book's instructions step by step, so in the end you can complete the project correctly.

CHAPTER 1

■ ■ ■

Introduction to Node.js

What Is Node.js?

Node.js is a low-level, highly scalable platform. It was created explicitly to be an experiment in asynchronous processing. Using Node.js, you will program directly with many network and Internet protocols or use libraries that have access to operating system (OS) resources. To program in Node.js you only need to master JavaScript language; that's right, only JavaScript! The JavaScript runtime used in this platform is the famous Javascript V8, which is also used in Google Chrome.

Main Features
Single-Thread Architecture

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

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, as the main process (or master process) works on balancing the load among the other processes (or slave process). If your server has multiple cores, applying this technique will optimize the usage of the processing.

Another way is to use asynchronous programming, which is one of the main resources from JavaScript. The asynchronous functions on Node.js work as *nonblocking I/O:* If your application has to read a huge file it will not block the CPU, because the I/O operation runs out of a thread pool, allowing your application to continue to process other tasks requested by other users.

1

Event-Loop

Node.js has an *event-oriented paradigm*. It follows the same event-oriented philosophy on the 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 stream, and many others.

The event-loop (Figure 1-1) is the agent responsible for noticing and emitting events. Simply speaking, it is an infinite loop that in each iteration verifies in the event queue if some event was triggered. When an event is triggered, the event-loop executes it and sends 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.
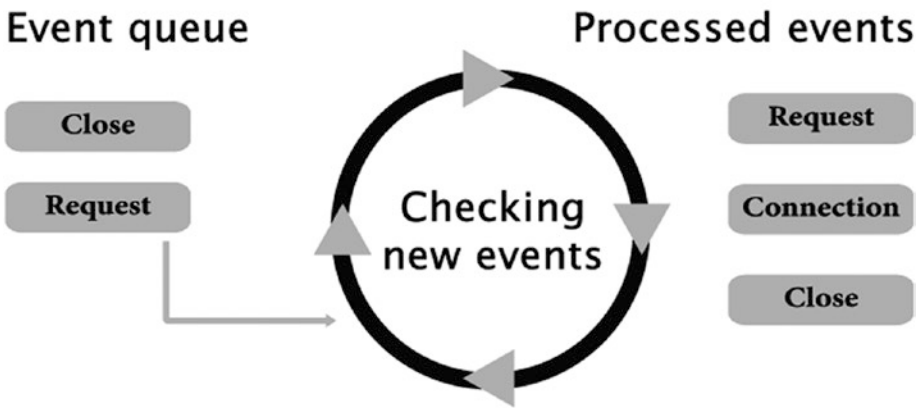
Figure 1-1. The event-loop process

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 streamline your learning process; after all, the language is the same as the JavaScript client side. Your main challenge in this platform is to learn how to work with the asynchronous programming to make the most of this technique. Another advantage of working with JavaScript is that you can keep a project that is easy to maintain, as long as you know how 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 specific frameworks to parse JavaScript Object Notation (JSON) objects; after all, the JSON from the client side will be same as on the server side. There are also cases of applications running databases that persist JSON objects, such as the popular NoSQL: MongoDB, CouchDB, and Riak. An important detail is that Node.js uses many functionalities from ECMAScript 6, allowing you write 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 at events, or publishing and maintaining new modules.

- *Active open source modules*: Today there are more than 300,000 open source modules published into NPM that can be used instantly for Node.js projects.

- *Ready to real time*: Node.js has become popular thanks to some frameworks for real-time communications between client and server. SockJS and Socket.IO are good examples. They are compatible with the WebSockets protocol and they allow you to traffic data through a single bidirectional connection, treating all data by JavaScript events.

- *Big players*: LinkedIn, WalMart, 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, I explained the main concepts and advantages of using Node.js. In the next chapters, we are going to see how it works in practice, with a single condition: Open your mind to new ideas and read this book with excitement

CHAPTER 2

■ ■ ■

Setting Up the Environment

In this chapter, we are going to install Node.js on the most used OSs (Windows, Linux, and MacOSX). However, the entire book uses MacOSX as the default OS. Don't worry about the differences among these OSs, because all examples in this book are compatible with all the three OS platforms.

Node.js Standard Installation

Setting up the Node.js environment is much the same despite the OS. Just a few procedures will be different in each OS, especially for Linux, because in the Linux environment you can compile an application or install it using a package manager, but this is not a major change.

The first step is to access the official Node.js web site at `http://nodejs.org`. On the home page, shown in Figure 2-1, v6.9.1 LTS Stable to download the latest compatible version for Windows or MacOSX. If you use Linux, I recommend you read the quick wiki at `https://github.com/nodejs/node/wiki/Installing-and-Building-Node.js`, which explains how to compile and install Node.js in some popular Linux distros.

Figure 2-1. *Node.js home page*

After the download, install the software. If you use Windows or MacOSX, just click through the installation wizard screens until the installation is completed; after all, there is no specific configuration required.

To test if everything is running correctly, just open the terminal (for Linux or MacOSX), command prompt, or Power Shell (for Windows) and type this command:

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

This checks the version of Node.js and NPM. This book uses Node v6.9.1 LTS and NPM 3.10.8**.**

About io.js and Node.js Merge

As of September 2015, Node.js was upgraded from 0.12.x to version 4.0.0 because of the merge with 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, in addition to making several improvements that were slow on Node.js. When io.js reached version 3.0.0 and both groups decided to merge io.js into Node.js, the result was the Node.js v4.0.0. This merger not only gave a huge upgrade to this platform, but also resulted in a more stable and reliable version of Node.js.

■ **Note**   This book uses Node v6.9.1 LTS**,** with a lot of ES6 codes. You can learn more about ECMAScript 6 at `http://es6-features.org`.

Node.js Installation Via NVM

■ **Tip**   In this section, we are just going to explore an alternative way to set up Node. js using a version manager. Feel free to skip this step if you don't need to or don't want to learn how to use the NVM.

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

NVM is a perfect solution to test your projects in different Node.js versions. It is also useful for people who like to test unstable or the latest versions. NVM has great advantages: It is practical, it is easy to use, it can uninstall a Node.js version with a single command, and it will save you time searching and installing a versions of Node.js. It is a good alternative, especially on Linux, because its native package managers are frequently outdated, which invalidates a new Node.js version to be installed.

Set Up NVM

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

```
1   curl https://raw.githubusercontent.com/creationix/nvm/v0.32.1/install.
sh | bash
```

Unfortunately, the official version of NVM is not available for 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 command-line interface (CLI) inspired by NVM.

Top NVM Commands

As as a sort of recipe, here is a list of 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 Node.js versions available to download and install.

- `nvm install vX.X.X`: Downloads and installs a version.

- `nvm uninstall vX.X.X`: Uninstalls an existing version.

- `nvm use vX.X.X`: Chooses an existing version to use.

- `nvm alias default vX.X.X`: Chooses an existing version to be loaded by default in the OS boot time.

■ **Note**   Replace the `vX.X.X` with the Node.js version of your choice, such as `v6.9.1`.

Installing Node.js Via NVM

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

```
1   nvm install v6.9.1
2   nvm use v6.9.1
3   nvm alias default v6.9.1
```

After running these commands, Node.js will be installed and ready to use.

Conclusion

Congratulations! Now, besides having 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 Node Package Manager (NPM), 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, Node Package Manager (NPM). It has become so popular throughout the community that beginning with 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.

Nowadays, the NPM home page, shown in Figure 3-1, hosts more than 350k of modules created and there are 4 million developers and thousands of companies using this platform.
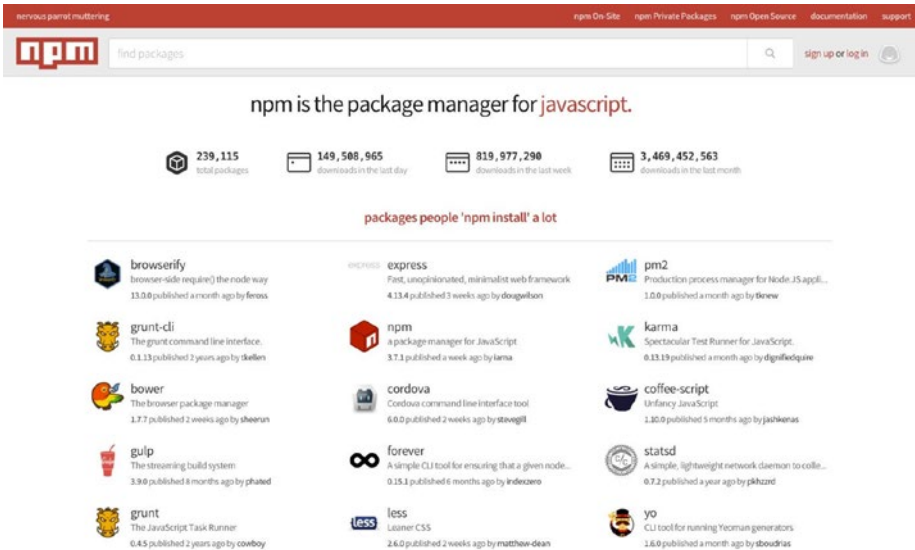
Figure 3-1. *The NPM home page at* `npmjs.org`

Top NPM Commands

NPM is very easy to use. You can use it to manage your project's dependencies, and when you create commands for task automation purposes, everything is created and managed into the `package.json` file. The following are the top NPM commands.

- `npm init`: Displays a simple wizard to help you create and describe your new project.

- `npm install module_name`: Installs a module.

- `npm install -g module_name`: Install a global module.

- `npm install module_name --save`: Installs a module and adds it into the `package.json` file, inside dependencies.

- `npm install module_name --save-dev`: Installs a module and adds 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`: Uninstalls a module from the project.

- `npm remove -g module_name`: Uninstalls a global module.

- `npm remove module_name --save`: Uninstalls a module from the project and also removes it from the attribute dependencies.

- `npm remove module_name --save-dev`: Uninstalls a module from the project and also removes it from the attribute devDependencies.

- `npm update module_name`: Updates the module version.

- `npm update -g module_name`: Updates the global module version.

- `npm -v`: Displays the NPM current version.

- `npm adduser username`: Creates an account to https://npmjs.org.

- `npm whoami`: Displays details of your public NPM profile (you must create an account using the previous command).

- `npm publish`: Publishes a module to npmjs.org (it's necessary to have an active account first).

Understanding the package.json File

All Node.js projects are called *modules.* What is a module? Throughout the book, note I use the terms *module, library,* and *framework.* In practice, they all mean the same thing.

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

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

To demonstrate it, here is a simple example of a `package.json` file that describes the main attributes of a module:

```
1   {
2     "name": "my-first-node-app",
3     "description": "My first node app",
4     "author": "User <user@email.com>",
5     "version": "1.2.3",
6     "private": true,
7     "dependencies": {
8       "module-1": "1.0.0",
9       "module-2": "~1.0.0",
10      "module-3": ">=1.0.0"
11    },
12    "devDependencies": {
13      "module-4": "*"
14    }
15  }
```

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

The `author` attribute is provides the name and e-mail of the author. You can use the format `Name <email>` so that web sites, such as `npmjs.org`, can read this information correctly.

Another main attribute is the `version`, which defines the current version of the module. It is highly recommended that you use this attribute to allow a module installation via the `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 a private project.

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

1. Major: X.0.0

2. Minor: 0.X.0

3. Patch: 0.0.X

Note the X means the current level version to date.

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

The next module, `"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 updates the x version). Generally, these updates are safe, as they bring improvements and bug fixes.

The `"module-3"` updates versions that are greater than 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, `"module-4",` uses the `"*"` character. This one always catches the latest version of the module in any version level. It also can cause trouble in the updates and it has the same behavior as the `module-3` versioning. Generally, it is used only in the `devDependencies`, which are dependencies used for test purposes that do not affect the application behavior. Do not use this kind of version in a production environment!

If you want to have 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 versions within the file `npm-shrinkwrap.json` giving you control over the dependencies versions of your project. This command is very useful in a production 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` as in this example.

```
1   {
2     "name": "my-first-node-app",
3     "description": "My first node app",
4     "author": "User <user@email.com>",
5     "version": "1.2.3",
6     "private": true,
7     "scripts": {
8       "start": "node app.js",
9       "clean": "rm -rf node_modules",
10      "test": "node test.js"
11    },
12    "dependencies": {
13      "module-1": "1.0.0",
14      "module-2": "~1.0.0",
15      "module-3": ">=1.0.0"
16    },
17    "devDependencies": {
18      "module-4": "*"
19    }
20  }
```

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 a shortcut, only the `start` and `test` commands can be executed by the alias: `npm start` and `npm test`. Within the scripts, you can run the command `node`, `npm` or any other global command from the OS. The `npm run clean` command is an example of a global command from the `bash`, which internally runs the command `rm -rf node_modules` to delete all files from the `node_modules` folder.

Conclusion

At this point, you have learned the basics about managing dependencies and automating tasks using NPM. It is important to understand these concepts because they are used frequently 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 a spoiler, I am going to tell you what this project will be.

In the next chapters, we are going to create a simple API REST system to manage tasks. In addition to the API, we'll also create a web application to consume data from this API. All development is going to happen using popular Node.js frameworks: Express to create the API resources, Sequelize to persist data in a relational database, Passport to deal with user authentications, and many more.

CHAPTER 4

■ ■ ■

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 has the advantage of an application focused only on feeding any client-side application with data. Nowadays, it's quite common to create web and mobile apps that consume data from one more APIs. This means that several kinds of client applications consult the same server focused on dealing with data. Besides, it also allows each application—client or server—to be worked with by different teams.

First, we are going to build an API. Throughout to the last chapters, however, we will also build a simple web client-side application to consume data from the 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 by the Sinatra framework from the Ruby language. With this module, you can create anything from small applications to large, complex ones. This framework allows you to build APIs and also to create simple web sites. The home page for Express is shown in Figure 4-1.
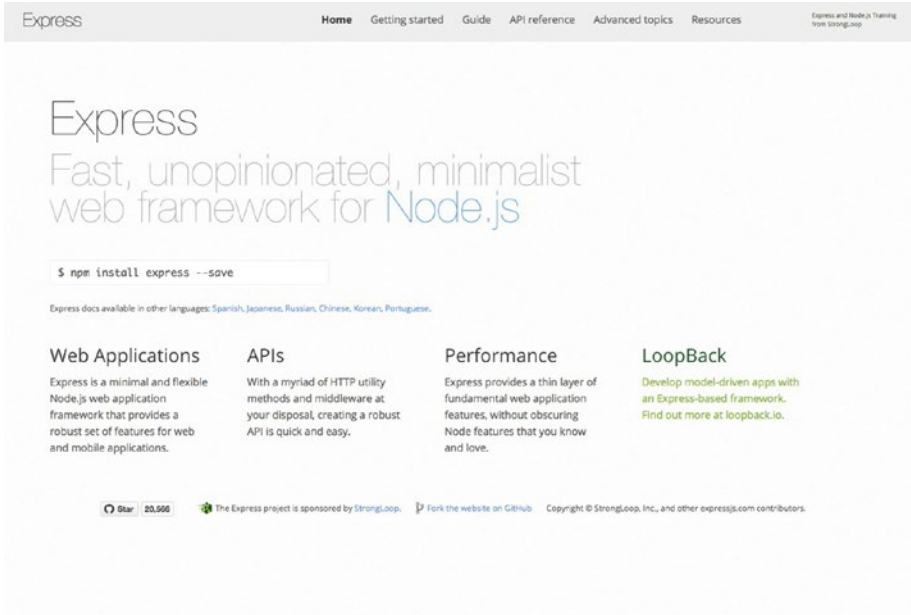
Figure 4-1. *The Express home page at* `expressjs.com`

It is focused on working with `views`, `routes`, and `controllers`, only `models` are 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 an important advantage, because there are many object data mappers (ODMs) and also object relational mapping (ORMs) available. You can use any of them with Express without a problem; you just need to load this kind of module, write some models, and use them inside the `controllers`, `routes`, or `views`.

Express allows developers to freely arrange the project's code; that is, there are 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 it's not always necessary to apply many conventions to 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 for some of their technical features. This independence forces the developer to fully understand how each application structure works.

To sum up, here is a list of the main features of Express.

- Robust routing.

- Easily integratable with a lot of template engines.

- Minimalist code.

- Works with middlewares concept.

- A huge list of third-party middlewares to integrate.

- Content negotiation.

- Adopt standards and best practices of REST APIs.

Getting Started on the Pilot Project

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

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

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

- List of tasks.

- Create, delete, and update a task.

- Create, delete, and update user data.

- User authentication.

- API documentation page.

Pilot Project Source Code
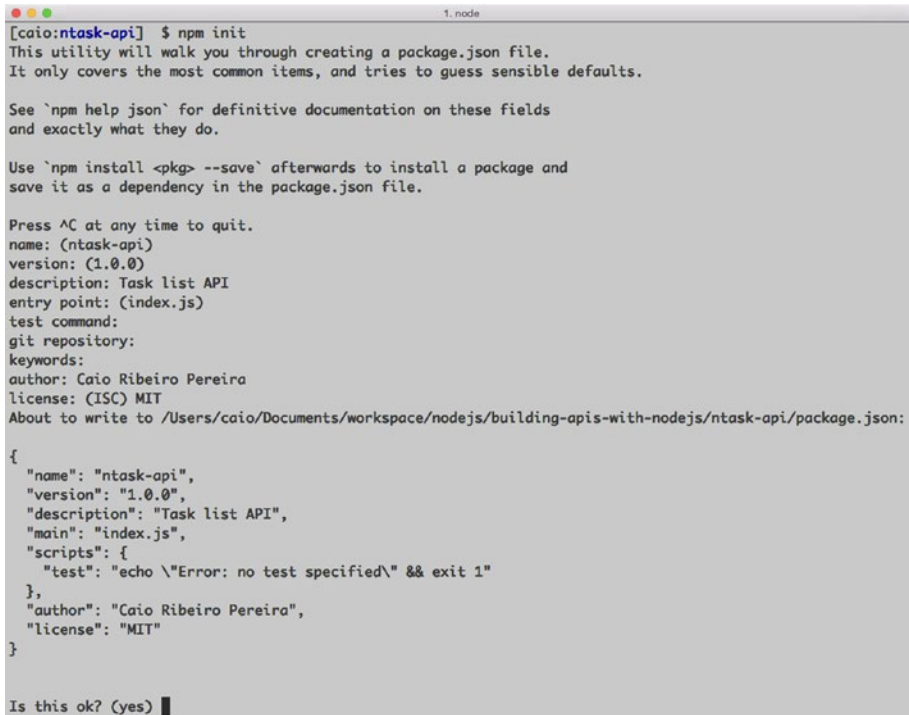
If you are curious to check out the source code of all projects that are going to be explored in this book, you can access it online at https://github.com/caio-ribeiro-pereira/building-apis-with-nodejs.

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

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

```
1   mkdir ntask-api
2   cd ntask-api
3   npm init
```

You need to answer the `npm init` command wizard as you like or you can copy the result shown in Figure 4-2.

Figure 4-2. *Describing the project using* `npm init`

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

```
1   {
2     "name": "ntask-api",
3     "version": "1.0.0",
4     "description": "Task list API",
5     "main": "index.js",
6     "scripts": {
7       "test": "echo \"Error: no test specified\" && exit 1"
8     },
9     "author": "Caio Ribeiro Pereira",
10    "license": "ISC"
11  }
```

The current 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 better. To do this, we are going to install `babel`, a JavaScript compiler responsible for converting ES6/ES7 codes to an 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 the `babel-cli` and `babel-preset-es2015` modules in the project, running this command:

```
1    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:

```
1    {
2      "presets": ["es2015"]
3    }
```

After that, we'll turbocharge `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 for starting our API server via the Babel compiler using the command `babel-node index.js`. Our `package.json` is going to look like this:

```
1    {
2      "name": "ntask-api",
3      "version": "1.0.0",
4      "description": "Task list API",
5      "main": "index.js",
6      "scripts": {
7        "start": "babel-node index.js"
8      },
9      "author": "Caio Ribeiro Pereira",
10     "dependencies": {
11       "babel-cli": "^6.5.1",
12       "babel-preset-es2015": "^6.5.0"
13     }
14   }
```

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

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

By 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 on the port 3000 using the function `app.listen()`. To do this, create the `index.js` file using this code:

```
1    import express from "express";
2
3    const PORT = 3000;
4    const app = express();
5
```

19
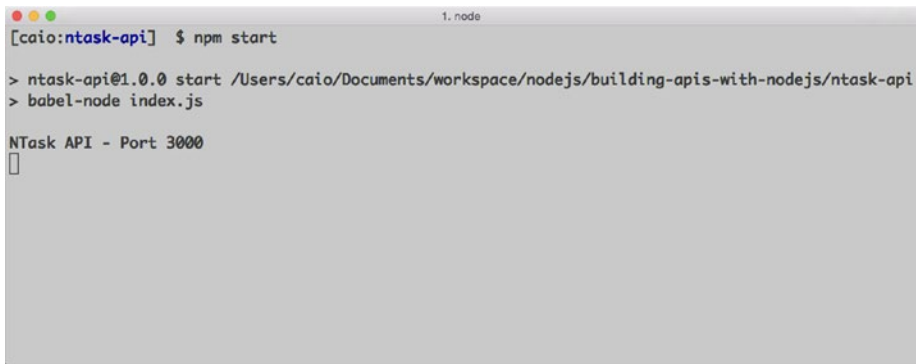
```
6    app.get("/", (req, res) => res.json({status: "NTask API"}));
7
8    app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

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

```
1    npm start
```

Your application must display the message shown in Figure 4-3.
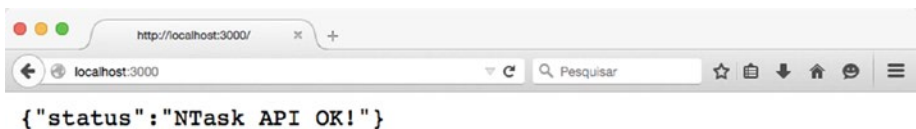
Figure 4-3. *Starting the API*

Next, open your browser and navigate to localhost:3000. If nothing goes wrong, a JSON status message will be displayed, similar to Figure 4-4.

Figure 4-4. *JSON status message*

Implementing a Simple and Static Resource

RESTful APIs work with the concept of creating and manipulating *resources*. These resources are entities that are used for queries, entries, and updating and deleting data, and everything is based on manipulating data from resources.

For example, our application will have as its 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 the same line as data modeling; the only difference is that the resources abstract the data source. Therefore, a resource can return data from different sources, such as databases, static data, or any data from external systems, and it can return data 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 perform some refactoring to integrate a database.

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

```
1   import express from "express";
2
3   const PORT = 3000;
4   const app = express();
5
6   app.get("/", (req, res) => res.json({status: "NTask API"}));
7
8   app.get("/tasks", (req, res) => {
9     res.json({
10      tasks: [
11        {title: "Buy some shoes"},
12        {title: "Fix notebook"}
13      ]
14    });
15  });
16
17  app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

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

■ **Caution**   Always follow this same step to restart the application correctly.

Now we have a new endpoint available to access via the address `localhost:3000/tasks`. Once you open it, you will have the result shown in Figure 4-5.

Figure 4-5. *Listing tasks*

If you want your results to return as formatted and tabbed JSON output, include in index.js the following configuration: app.set("json spaces", 4); as shown here.

```
1    import express from "express";
2
3    const PORT = 3000;
4    const app = express();
5
6    app.set("json spaces", 4);
7
8    app.get("/", (req, res) => res.json({status: "NTask API"}));
9
10   app.get("/tasks", (req, res) => {
11     res.json({
12       tasks: [
13         {title: "Buy some shoes"},
14         {title: "Fix notebook"}
15       ]
16     });
17   });
18
19   app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

Now, restart the server to see the more elegant result shown in Figure 4-6.
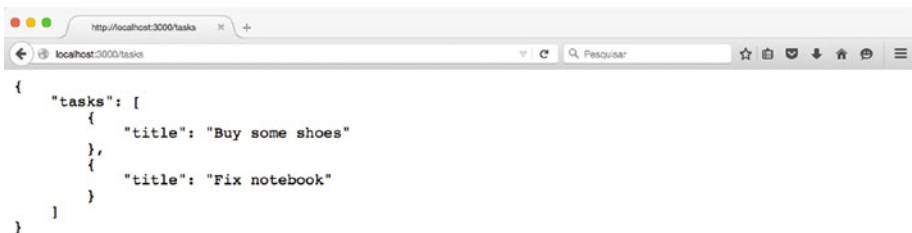
Figure 4-6. *Listing tasks with formatted JSON*

Arranging the Loading of Modules

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

We are going to apply the Model-View-Router (MVR) pattern to arrange this. To do it, we'll use the consign module, which will allow our project to autoload models, routers, middlewares, configs, and more, as this module injects dependencies easily. Let's install.

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

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

```
1   module.exports = app => {
2     app.get("/", (req, res) => {
3       res.json({status: "NTask API"});
4     });
5   };
```

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

```
1    module.exports = app => {
2      app.get("/tasks", (req, res) => {
3        res.json({
4          tasks: [
5            {title: "Buy some shoes"},
6            {title: "Fix notebook"}
7          ]
8        });
9      });
10   };
```

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

```
1    import express from "express";
2    import consign from "consign";
3
4    const PORT = 3000;
5    const app = express();
6
7    app.set("json spaces", 4);
8
9    consign()
10     .include("routes")
11     .into(app);
12
13   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 the MVR pattern. In our case, the JSON outputs are considered views, which are provided by the routes. The next step will be arranging the models.

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

```
1   import express from "express";
2   import consign from "consign";
3
4   const PORT = 3000;
5   const app = express();
6
7   app.set("json spaces", 4);
8
9   consign()
10    .include("models")
11    .then("routes")
12    .into(app);
13
14  app.listen(PORT, () => console.log(`NTask API - Port ${PORT}`));
```

At this moment, the `consign()` function won't load any model, because the directory `models` doesn't exist. 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 fill it with these codes.

```
1   module.exports = app => {
2     return {
3       findAll: (params, callback) => {
4         return callback([
5           {title: "Buy some shoes"},
6           {title: "Fix notebook"}
7         ]);
8       }
9     };
10  };
```

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 here, but it serves as a base to send some SQL query filters, something that we are going to cover in detail 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 the app variable. After all, the modules added by the `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 and edit the `routes/tasks.js` following this code.

```
1   module.exports = app => {
2     const Tasks = app.models.tasks;
3     app.get("/tasks", (req, res) => {
4       Tasks.findAll({}, (tasks) => {
5         res.json({tasks: tasks});
6       });
7     });
8   };
```

Note that the function `Tasks.findAll()` has in the first parameter an empty object. This is the value of the `params` variable. 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 some static values from his model. So, at this point, we are sure that `tasks` is going to return a static array with some task descriptions.

To finish these refactorings, let's create a file that is going to load all the middlewares and specific settings of Express. Currently, we only have a simple configuration of JSON's format, which occurs via the function `app.set("json spaces", 4)`. However, we are going to include another setting, in this case, 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, here I recommend preparing the house for these new visitors.

Create the file `libs/middlewares.js` and write this code.

```
1   module.exports = app => {
2     app.set("port", 3000);
3     app.set("json spaces", 4);
4   };
```
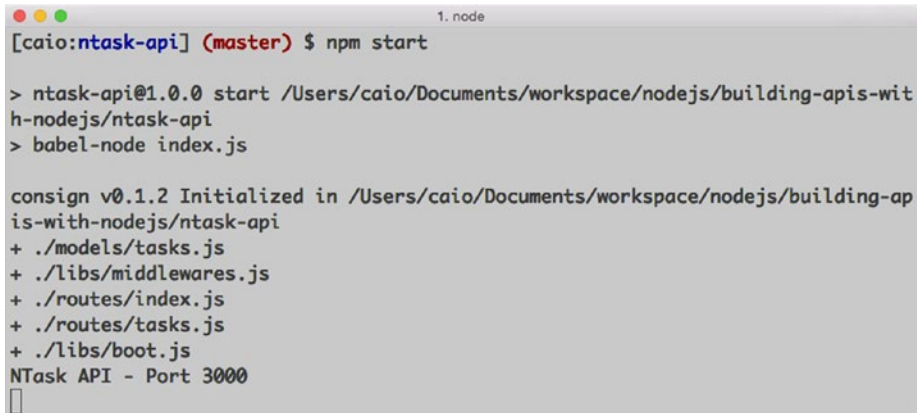
To simplify the server boot, create the file `libs/boot.js`, which is going to be responsible for the server initialization via the `app.listen()` function. This time, we'll use `app.get("port")` instead of the old constant PORT.

```
1   module.exports = app => {
2     app.listen(app.get("port"), () => {
3       console.log(`NTask API - Port ${app.get("port")}`);
4     });
5   };
```

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`.

```
1    import express from "express";
2    import consign from "consign";
3
4    const app = express();
5
6    consign()
7      .include("models")
8      .then("libs/middlewares.js")
9      .then("routes")
10     .then("libs/boot.js")
11     .into(app);
```

To test everything, restart the server and access the endpoint again: `localhost:3000/tasks`. To make sure everything is okay, no errors should occur and all the data must be displayed normally, as shown in Figure 4-7.

Figure 4-7. *Listing loaded modules*

Conclusion

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

CHAPTER 5

■ ■ ■

Working with SQL Databases

In Chapter 4, we created simple routes for the application to list tasks via a static data model. That was enough to start to explore some basic concepts about API resources.

Now, we are going to work deeper on using a relational database to create dynamic management of tasks and users. To simplify our examples, we are going to use SQLite3 (Figure 5-1). It is a preinstalled database on Linux, Unix, and MacOSX, so there's no need to set it up. However, if you are using Windows, you can easily install it by following the instructions at https://www.sqlite.org/download.htm.

Figure 5-1. *SQLite3 logo*

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

On Node.js, there are several frameworks that work with SQLite3. On our application, we'll use the Sequelize.js module (Figure 5-2), which is a complete module that has a nice, user-friendly interface. 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.

Figure 5-2. *Sequelize.js logo*

Sequelize is an ORM for Node.js. It is a Promises-based framework. Currently, it is on version 3.19.0 and has a lot of features like transactions, modeling tables, table relationships, database replication for reading mode, and more.

To read the full documentation, visit the Sequelize home page shown in Figure 5-3 at http://sequelizejs.com.
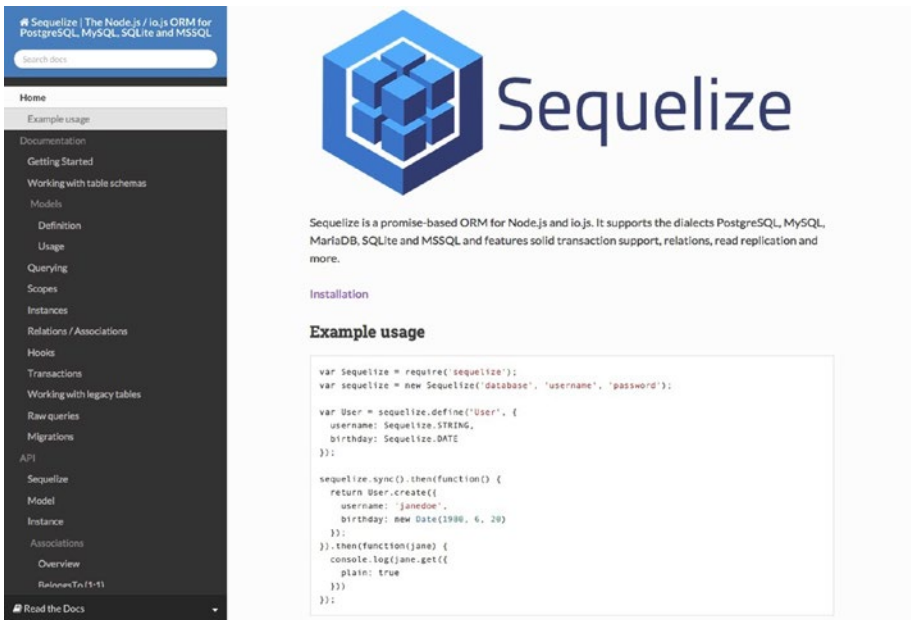
Figure 5-3. *Sequelize.js home page*

Setting Up Sequelize

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

```
1   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 so, 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`: A specific attribute for only SQLite3, it defines the directory where the database files will be recorded.

- `params.define.underscored`: Standardizes the tables fields' names to appear in lowercase letters with an underscore.

The file is going to look like this.

```
1   module.exports = {
2     database: "ntask",
3     username: "",
4     password: "",
5     params: {
6       dialect: "sqlite",
7       storage: "ntask.sqlite",
8       define: {
9         underscored: true
10      }
11    }
12  };
```

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

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

```
1   import Sequelize from "sequelize";
2   const config = require("./libs/config.js");
3   let sequelize = null;
4
5   module.exports = () => {
6     if (!sequelize) {
7       sequelize = new Sequelize(
8         config.database,
9         config.username,
10        config.password,
11        config.params
12      );
13    }
14    return sequelize;
15  };
```

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

```
1   import express from "express";
2   import consign from "consign";
3
4   const app = express();
5
6   consign()
7     .include("db.js")
8     .then("models")
9     .then("libs/middlewares.js")
10    .then("routes")
11    .then("libs/boot.js")
12    .into(app);
```

To finish the Sequelize.js setup, let's implement a simple sync function between Sequelize and the database. This sync function performs, if necessary, alterations on database tables, according to what is going to be set up 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 the libs/boot.js file.

```
1   module.exports = app => {
2     app.db.sync().done(() => {
3       app.listen(app.get("port"), () => {
4         console.log(`NTask API - Port ${app.get("port")}`);
5       });
6     });
7   };
```

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

Creating Models

To this point, the only model in the API is `model/tasks.js`, which is a static model that is not connected with any database. In this section, we explore the main Sequelize features to create models that perform 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 Figure 5-4.
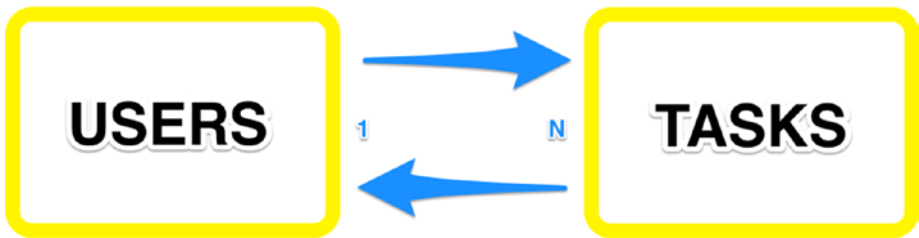
Figure 5-4. *NTask models relationship*

To work with this kind of relationship, we use 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 in the model. In our case, we are going to create the `associate` function within the `classMethods` of each model.

Model: Tasks

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

```
1   module.exports = (sequelize, DataType) => {
2     const Tasks = sequelize.define("Tasks", {
3       id: {
4         type: DataType.INTEGER,
5         primaryKey: true,
6         autoIncrement: true
7       },
8       title: {
9         type: DataType.STRING,
10        allowNull: false,
```

```
11          validate: {
12            notEmpty: true
13          }
14        },
15        done: {
16          type: DataType.BOOLEAN,
17          allowNull: false,
18          defaultValue: false
19        }
20      }, {
21        classMethods: {
22          associate: (models) => {
23            Tasks.belongsTo(models.Users);
24          }
25        }
26      });
27      return Tasks;
28    };
```

The function `sequelize.define("Tasks")` is responsible for creating or changing a table. This happens only when Sequelize syncs with the application during boot time (in our case, this happens on the `libs/boot.js` file via the `app.db.sync()` function). The second parameter is an object; their attributes respectively represent the fields of a table, and their values are subattributes that describe the data type of 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 autoincremented (`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 it 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`) that 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 the `defaultValue: false` attribute.

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

■ **Warning**   The model `Users` hasn't been created yet, so if you restart the server right now, an error will occur. Stay calm, then keep reading and coding until the end of this chapter to see the final result.

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.

```
1   module.exports = (sequelize, DataType) => {
2     const Users = sequelize.define("Users", {
3       id: {
4         type: DataType.INTEGER,
5         primaryKey: true,
6         autoIncrement: true
7       },
8       name: {
9         type: DataType.STRING,
10       allowNull: false,
11       validate: {
12         notEmpty: true
13       }
14      },
15      password: {
16        type: DataType.STRING,
17        allowNull: false,
18        validate: {
19          notEmpty: true
20        }
21      },
22      email: {
23        type: DataType.STRING,
24        unique: true,
25        allowNull: false,
26        validate: {
27          notEmpty: true
28        }
29      }
30     }, {
31       classMethods: {
32         associate: (models) => {
33           Users.hasMany(models.Tasks);
34         }
35       }
36     });
37     return Users;
38  };
```

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 e-mails 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, let's modify some of the modules loading on index.js. First, we are going to order the loading of the modules so the libs/config.js is loaded first, then the db.js, and then also remove the loading of the models directory from consign, because now all models are loaded from db.js. The code should look like this.

```
1    import express from "express";
2    import consign from "consign";
3
4    const app = express();
5
6    consign()
7      .include("libs/config.js")
8      .then("db.js")
9      .then("libs/middlewares.js")
10     .then("routes")
11     .then("libs/boot.js")
12     .into(app);
```

The reason the directory `models` was deleted from `consign()` is that now, all models will be loaded directly by `db.js` file, via the `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. Let's refactor the `db.js` file, to import and load all models correctly.

```
1    import fs from "fs";
2    import path from "path";
3    import Sequelize from "sequelize";
4
5    let db = null;
6
7    module.exports = app => {
8      if (!db) {
9        const config = app.libs.config;
10       const sequelize = new Sequelize(
11         config.database,
12         config.username,
13         config.password,
14         config.params
15       );
16       db = {
17         sequelize,
18         Sequelize,
19         models: {}
20       };
21       const dir = path.join(__dirname, "models");
22       fs.readdirSync(dir).forEach(file => {
23         const modelDir = path.join(dir, file);
24         const model = sequelize.import(modelDir);
25         db.models[model.name] = model;
26       });
27       Object.keys(db.models).forEach(key => {
28         db.models[key].associate(db.models);
```

```
29     });
30     }
31   return db;
32   };
```

This time, the code got a little complex, didn't it? Its features are pretty cool, though. 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 an array of strings referring to the file 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 the `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()`:

```
1   module.exports = app => {
2     app.db.sequelize.sync().done(() => {
3       app.listen(app.get("port"), () => {
4         console.log(`NTask API - Port ${app.get("port")}`);
5       });
6     });
7   };
```

Then, let's modify `routes/tasks.js` to load the Tasks model correctly using `app.db.models.Tasks` and modify the function `Tasks.findAll()` to this Promises-based function provided by Sequelize. This is how it should look.

```
1   module.exports = app => {
2     const Tasks = app.db.models.Tasks;
3     app.get("/tasks", (req, res) => {
4       Tasks.findAll({}).then(tasks => {
5         res.json({tasks: tasks});
6       });
7     });
8   };
```

Conclusion

Finally we have finished the Sequelize.js adaptation into our API. To test it, just restart the server and go to http://localhost:3000/tasks.

This time it is going to display a JSON object with empty tasks, as shown in Figure 5-5.
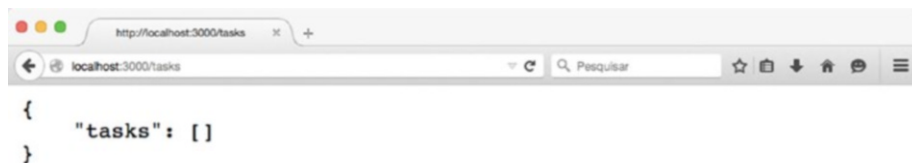
Figure 5-5. *Now, the tasks list is empty*

Don't worry, though: In the next chapter we'll implement the main endpoints to perform a complete CRUD in the API. Just keep reading!

■ ■ ■

CRUDify API Resources

In this chapter, we'll continue to explore the new functions from Sequelize and also organize the API's routes and some middlewares of Express. The create, read, update, delete (CRUD) will be built using the `Tasks` and `Users` models.

Organizing Task 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 us to use nested functions, reusing the routes through all HTTP methods provided by the following Express functions.

- `app.all()`: A middleware that is executed via any HTTP method call.

- `app.get()`: Executes the method `GET` from HTTP, and is used to search and get some set of data from one or multiple resources.

- `app.post()`: Executes the method `POST` from HTTP, and is semantically used to send and persist new data into a resource.

- `app.put()`: Executes the method `PUT` from HTTP, and is used to update the data of a resource.

- `app.patch()`: Executes the method `PATCH` to HTTP, and has similar semantics to the `PUT` method, but is recommended only to update some attributes of a resource, not the entire data of a resource.

- `app.delete()`: Executes the method `DELETE` from HTTP, as its name implies, to delete particular data of a resource.

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

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

```
 3
 4    app.route("/tasks")
 5      .all((req, res) => {
 6        // Middleware for preexecution of routes
 7      })
 8      .get((req, res) => {
 9        // "/tasks": List tasks
10      })
11      .post((req, res) => {
12        // "/tasks": Save new task
13      });
14
15    app.route("/tasks/:id")
16      .all((req, res) => {
17        // Middleware for preexecution of routes
18      })
19      .get((req, res) => {
20        // "/tasks/1": Find a task
21      })
22      .put((r eq, res) => {
23        // "/tasks/1": Update a task
24      })
25      .delete((req, res) => {
26        // "/tasks/1": Delete a task
27      });
28  };
```

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" and "/tasks/:id") we are going to deal with some rules inside their middlewares via the app.all() function to avoid some invalid access if somebody sends the task id inside the request's body. This is going to be a simple modification, which should look like this.

```
 1  app.route("/tasks")
 2    .all((req, res, next) => {
 3      delete req.body.id;
 4      next();
 5    })
 6    // Continuation of routes...
 7
 8  app.route("/tasks/:id")
 9    .all((req, res, next) => {
10      delete req.body.id;
11      next();
12    })
13    // 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 function, 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` for a task.

To finish the middleware, warning that it should perform a corresponding function to an HTTP method, you just need to include at the end of the 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, we'll use the `app.get()` function, which is going to list all data from `tasks` models, running the `Tasks.findAll()` function.

```
1   app.route("/tasks")
2     .all((req, res, next) => {
3       delete req.body.id;
4       next();
5     })
6     .get((req, res) => {
7       Tasks.findAll({})
8         .then(result => res.json(result))
9         .catch(error => {
10          res.status(412).json({msg: error.message});
11        });
12    })
```

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. Throughout the chapters that follow, however, some arguments will be implemented to make this list of tasks more specific. The search results occur via the `then()` function, and any problems that happen can be handled via the `catch()` function. An important detail about an 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 the `200 - OK` status by default. On our API, to simplify things, several common errors will use the `412 - Precondition Failed` status code.

About HTTP Status

There are no rules to follow in the HTTP status definition; however, it's advisable to understand the meaning of each status to make your server's response more semantic. To learn more about HTTP status codes, visit www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Creating Tasks Via POST

There is no secret in the implementation of the 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` has some attributes not defined in the model, they will be removed before inserts or updates are made to a model. The only problem is the `req.body.id` that could modify the `id` of a model. However, we already handled this when we wrote a simple rule into the `app.all()` middleware. The implementation of this route will be like this piece of code.

```
1   app.route("/tasks")
2     .all((req, res, next) => {
3       delete req.body.id;
4       next();
5     })
6     .get((req, res) => {
7       Tasks.findAll({})
8         .then(result => res.json(result))
9         .catch(error => {
10          res.status(412).json({msg: error.message});
11        });
12    })
13    .post((req, res) => {
14      Tasks.create(req.body)
15        .then(result => res.json(result))
16        .catch(error => {
17          res.status(412).json({msg: error.message});
18        });
19    });
```

Finding a Task Via GET

Now we are going to treat the /tasks/:id endpoints. To do this, let's start with the implementation of the `app.route("/tasks/:id")` function, which has 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 a single search of tasks based on the task id. If there isn't a task, the API will respond with the `404 - Not Found` status code via the `res.sendStatus(404)` function. It should look like this.

```
1   app.route("/tasks/:id")
2     .all((req, res, next) => {
3       delete req.body.id;
4       next();
5     })
```

```
 6      .get((req, res) => {
 7        Tasks.findOne({where: req.params})
 8          .then(result => {
 9            if (result) {
10              res.json(result);
11            } else {
12              res.sendStatus(404);
13            }
14          })
15          .catch(error => {
16            res.status(412).json({msg: error.message});
17          });
18      })
```

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 with a first parameter in which 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. This information won't be very useful as a response, though, 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 and no content will be returned in the response. This implementation looks like this.

```
 1  app.route("/tasks/:id")
 2    .all((req, res, next) => {
 3      delete req.body.id;
 4      next();
 5    })
 6    .get((req, res) => {
 7      Tasks.findOne({where: req.params})
 8        .then(result => {
 9          if (result) {
10            res.json(result);
11          } else {
12            res.sendStatus(404);
13          }
14        })
15        .catch(error => {
16          res.status(412).json({msg: error.message});
17        });
18    })
19    .put((req, res) => {
20      Tasks.update(req.body, {where: req.params})
21        .then(result => res.sendStatus(204))
22        .catch(error => {
```

```
23          res.status(412).json({msg: error.message});
24        });
25     })
```

Just like the `Tasks create()` function, the `Tasks.update` cleans the fields that are
not on its own model, so there is no problem sending `req.body` directly as a 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 the `res.sendStatus(204)` function:

```
1    app.route("/tasks/:id")
2      .all((req, res, next) => {
3        delete req.body.id;
4        next();
5      })
6      .get((req, res) => {
7        Tasks.findOne({where: req.params})
8          .then(result => {
9            if (result) {
10              res.json(result);
11            } else {
12              res.sendStatus(404);
13            }
14          })
15          .catch(error => {
16            res.status(412).json({msg: error.message});
17          });
18      })
19      .put((req, res) => {
20        Tasks.update(req.body, {where: req.params})
21          .then(result => res.sendStatus(204))
22          .catch(error => {
23            res.status(412).json({msg: error.message});
24          });
25      })
26      .delete((req, res) => {
27        Tasks.destroy({where: req.params})
28          .then(result => res.sendStatus(204))
29          .catch(error => {
30            res.status(412).json({msg: error.message});
31          });
32      });
```

So, we have finish to CRUDifying 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 the `app.all()` function to an Express middleware, using the `app.use()` function, into the `lib-s/middlewares.js` file. To apply this refactoring, first, remove the function `app.all()` from the `routes/tasks.js` file, leaving the code with the following structure:

```
1   module.exports = app => {
2     const Tasks = app.db.models.Tasks;
3
4     app.route("/tasks")
5       .get((req, res) => {
6         // GET /tasks callback...
7       })
8       .post((req, res) => {
9         // POST /tasks callback...
10      });
11
12     app.route("/tasks/:id")
13       .get((req, res) => {
14         // GET /tasks/id callback...
15       })
16       .put((req, res) => {
17         // PUT /tasks/id callback...
18       })
19       .delete((req, res) => {
20         // DELETE /tasks/id callback...
21       });
22   };
```

To enable a JSON parse inside all the API's routes, we must install the `body-parser` moduleby running this command.

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

After this installation, open and edit the `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 this code.

```
1   import bodyParser from "body-parser";
2
3   module.exports = app => {
4     app.set("port", 3000);
5     app.set("json spaces", 4);
```

```
6      app.use(bodyParser.json());
7      app.use((req, res, next) => {
8        delete req.body.id;
9        next();
10     });
11   };
```

Creating Users' Endpoints

We also have to create routes to manage users in the application. After all, without them, it becomes impossible to manage tasks.

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

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

To code the users' routes, creates the routes/users.js file, using this code.

```
1    module.exports = app => {
2      const Users = app.db.models.Users;
3
4      app.get("/users/:id", (req, res) => {
5        Users.findById(req.params.id, {
6          attributes: ["id", "name", "email"]
7        })
8        .then(result => res.json(result))
9        .catch(error => {
10         res.status(412).json({msg: error.message});
11       });
12     });
13
14     app.delete("/users/:id", (req, res) => {
15       Users.destroy({where: {id: req.params.id} })
16         .then(result => res.sendStatus(204))
17         .catch(error => {
18           res.status(412).json({msg: error.message});
19         });
20     });
21
22     app.post("/users", (req, res) => {
23       Users.create(req.body)
```

```
24          .then(result => res.json(result))
25          .catch(error => {CRUDify API resources 45
26            res.status(412).json({msg: error.message});
27          });
28      });
29    };
```

■ **Note**   The reason 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 into the API.

Testing Endpoint 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 methods. To simplify, I recommend you use Postman, which is a useful Google Chrome extension that is easy to use. To install it go to `http:///www.getpostman.com`.

After the installation, open the Chrome Apps page and click the Postman icon, shown in Figure 6-1.
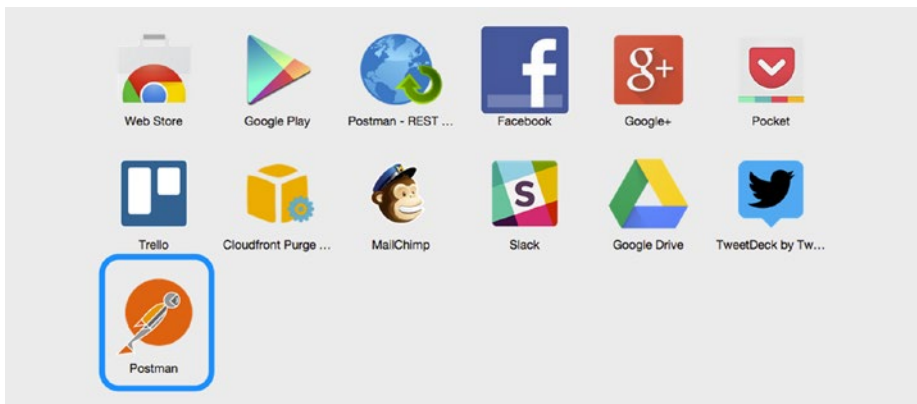
Figure 6-1. *Postman REST client app*

A login page is displayed, as shown in Figure 6-2, but you do not need to log in to use it. To skip this step and go straight to the main page, click Go to the app.
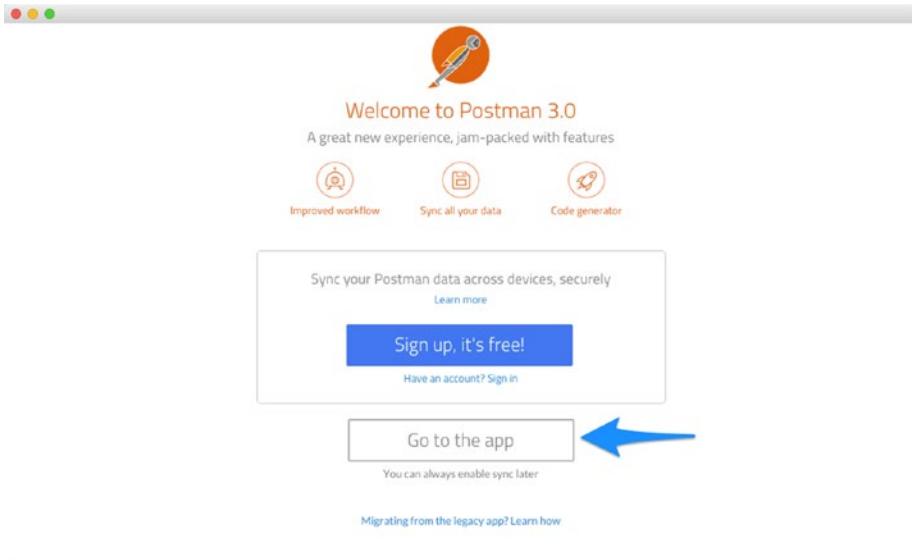
Figure 6-2. *Opening Postman*

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

1. Choose the method POST via the address
   http://localhost:3000/tasks.

2. Click on the menu Body, choose the option raw, and change
   the format from Text to JSON (application/json).

3. Create the JSON: {"title": "Sleep"} and click Send
   (see Figure 6-3).

4. Modify the same JSON to {"title": "Study"} and click Send
   again (see Figure 6-4).
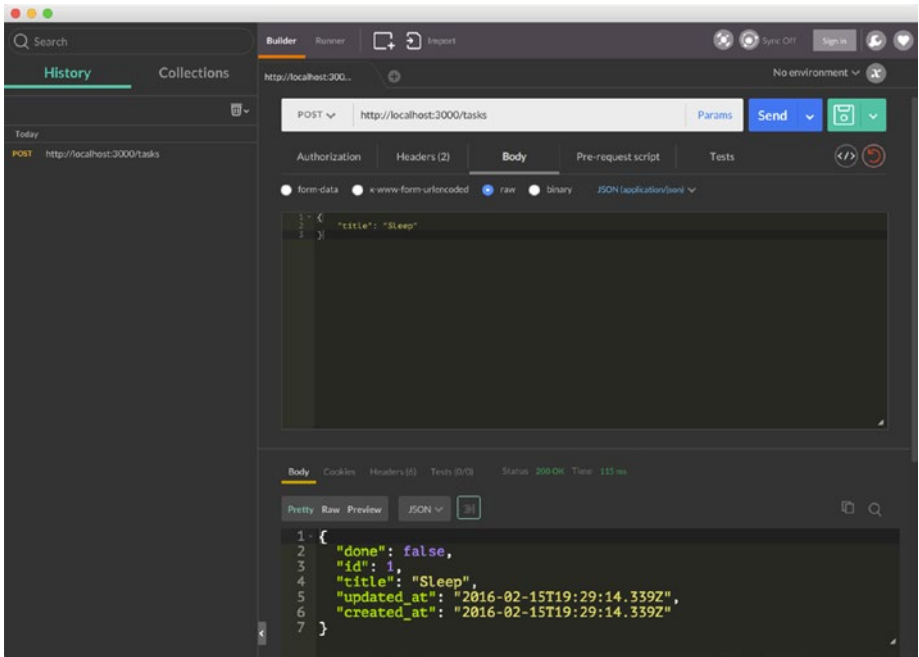
Figure 6-3. *Registering the task "Sleep"*
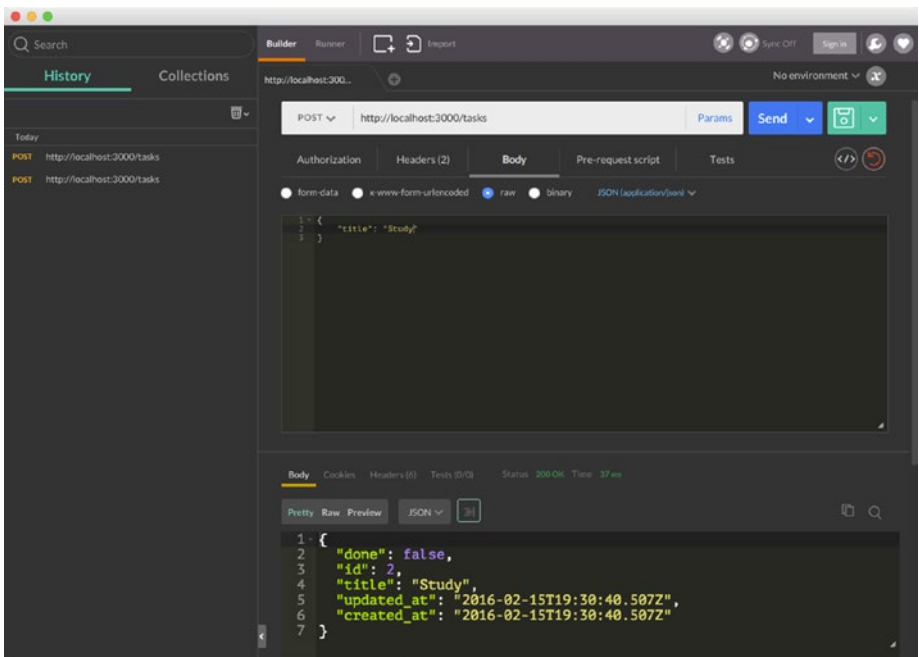
Figure 6-4. *Registering the task "Study"*

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`

- Method `GET`, route `http://localhost:3000/tasks/1`

- Method `GET`, route `http://localhost:3000/tasks/2`

- Method `PUT`, route `http://localhost:3000/tasks/1` using the body `{"title": "Work"}`

- Method `DELETE`, route `http://localhost:3000/tasks/2`

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

Conclusion

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

■ ■ ■

Authenticating Users

Our API already has tasks and users resources, which, thanks to the Sequelize framework, are integrated into 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 explore the main concepts and implementations of user 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 very cool Node.js module that is easy to work with for user authentication 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. It also allows you to work with external services authentication as single sign-on by using existing social networking account information, such as Facebook, Google+, Twitter, and more. By the way, on the official Passport web site at `http://passportjs.org` (see Figure 7-1), there is a list with more than 300 authentication strategies created and maintained by third parties.
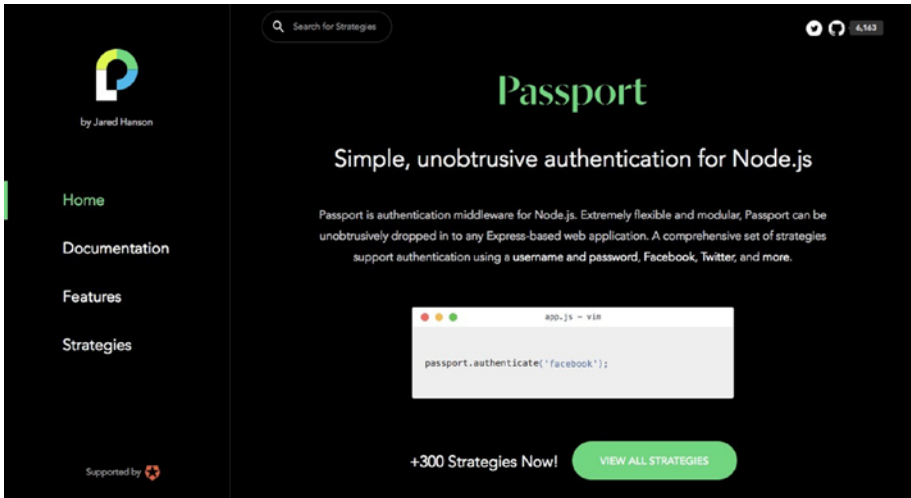
Figure 7-1. *Passport home page*

About JWT

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

1. Client makes a request once by sending 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. Optionally, this token could have a expiration date, to enforce the authentication's security.

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. Server always validates this token to allow or deny a customer request.

For specific details about JWT, go to `http://jwt.io` (see Figure 7-2).
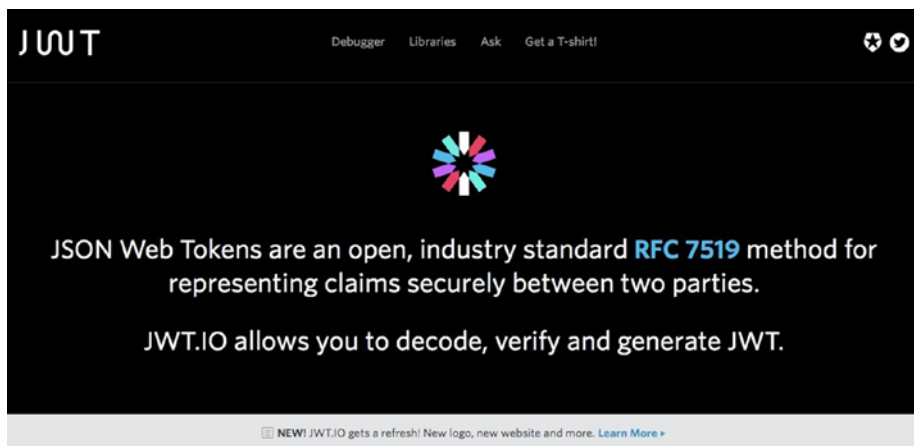
Figure 7-2. *JWT home page*

Installing Passport and JWT

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

- *Passport*: This will be used as the authentication engine.

- *Passport JWT*: This is the JWT authentication strategy for Passport.

- *JWT Simple*: This is used as encoder and decoder JSON tokens.

Now, let's install them by running this command:

```
1   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:

```
1   module.exports = {
2     database: "ntask",
3     username: "",
4     password: "",
5     params: {
6       dialect: "sqlite",
7       storage: "ntask.sqlite",
8       define: {
9         underscored: true
10       }
11     },
```

```
12     jwtSecret: "Nta$K-AP1",
13     jwtSession: {session: false}
14   };
```

The field jwtSecret keeps a secret key string that serves as a base to encode and decode tokens. It's highly advisable to use a complex string that uses many different characters. Never share or publish this secret key in public, because if it leaks, you will leave your application vulnerable, making it possible for those with bad intentions to access the system and manage the tokens from logged users without using the correct credentials in the system.

To finish, the last included field is jwtSession, which has the value {session: false}. This item is going to be used to inform Passport that the API won't manage the 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 when it starts the application, and it basically receives in this callback a payload that contains a decoded JSON that 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 an argument for the 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:

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

This middleware will be injected via the 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 Passport, and authenticate(), which is used to authenticate the access for a route.

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

```
1   import passport from "passport";
2   import {Strategy, ExtractJwt} from "passport-jwt";
3
4   module.exports = app => {
5     const Users = app.db.models.Users;
6     const cfg = app.libs.config;
7     const params = {
8       secretOrKey: cfg.jwtSecret,
9       jwtFromRequest: ExtractJwt.fromAuthHeader()
10    };
11    const strategy = new Strategy(params, (payload, done) => {
12        Users.findById(payload.id)
```

```
13              .then(user => {
14                if (user) {
15                  return done(null, {
16                    id: user.id,
17                    email: user.email
18                  });
19                }
20                return done(null, false);
21              })
22                .catch(error => done(error, null));
23          });
24      passport.use(strategy);
25      return {
26        initialize: () => {
27          return passport.initialize();
28        },
29        authenticate: () => {
30          return passport.authenticate("jwt", cfg.jwtSession);
31        }
32      };
33    };
```

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

```
1    import express from "express";
2    import consign from "consign";
3
4    const app = express();
5
6    consign()
7      .include("libs/config.js")
8      .then("db.js")
9      .then("auth.js")
10     .then("libs/middlewares.js")
11     .then("routes")
12     .then("libs/boot.js")
13     .into(app);
```

To initiate the Passport middleware, edit the libs/middlewares.js and include the middleware app.use(app.auth.initialize()). The following code shows where to include it.

```
1    import bodyParser from "body-parser";
2
3    module.exports = app => {
4      app.set("port", 3000);
5      app.set("json spaces", 4);
6      app.use(bodyParser.json());
```

```
 7     app.use(app.auth.initialize());
 8     app.use((req, res, next) => {
 9       delete req.body.id;
10       next();
11     });
12   };
```

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 for 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 uses the id of an authenticated user. Doing this, we complete 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 this command.

```
1   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, we will include 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 saving it.

A new function inside the classMethods will be included as well. It is 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.

```
 1   import bcrypt from "bcrypt";
 2
 3   module.exports = (sequelize, DataType) => {
 4     const Users = sequelize.define("Users", {
 5       // Users fields, defined in Chapter 5...
 6     }, {
 7       hooks: {
 8         beforeCreate: user => {
 9           const salt = bcrypt.genSaltSync();
10           user.password = bcrypt.hashSync(user.password, salt);
11         }
12       },
13       classMethods: {
14         associate: models => {
15           Users.hasMany(models.Tasks);
16         },
17         isPassword: (encodedPassword, password) => {
18           return bcrypt.compareSync(password, encodedPassword);
19         }
```

```
20        }
21      });
22      return Users;
23    };
```

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 and req.body.password.

The payload is going to have only the user id. The token generation occurs via the jwt-simple module using the function jwt.encode(payload, cfg.jwtSecret) that must use the same secret key jwtSecret that 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 token generation, you need to create the routes/token.js file using the following code.

```
1   import jwt from "jwt-simple";
2
3   module.exports = app => {
4     const cfg = app.libs.config;
5     const Users = app.db.models.Users;
6     app.post("/token", (req, res) => {
7       if (req.body.email && req.body.password) {
8         const email = req.body.email;
9         const password = req.body.password;
10        Users.findOne({where: {email: email}})
11          .then(user => {
12            if (Users.isPassword(user.password, password)) {
13              const payload = {id: user.id};
14              res.json({
15                token: jwt.encode(payload, cfg.jwtSecret)
16              });
17            } else {
18              res.sendStatus(401);
19            }
20          })
21          .catch(error => res.sendStatus(401));
22      } else {
23        res.sendStatus(401);
24      }
25    });
26  };
```

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 validate the tokens sent by clients and allow (or deny) access in some routes. To do this, edit the routes/tasks.js

file and add the middleware function `all(app.auth.authenticate())` at the beginning of both endpoints. The code should look like this.

```
1   module.exports = app => {
2     const Tasks = app.db.models.Tasks;
3
4     app.route("/tasks")
5       .all(app.auth.authenticate())
6       .get((req, res) => {
7         // "/tasks": List tasks
8       })
9       .post((req, res) => {
10        // "/tasks": Save new task
11      });
12
13    app.route("/tasks/:id")
14      .all(app.auth.authenticate())
15      .get((req, res) => {
16        // "/tasks/1": Find a task
17      })
18      .put((req, res) => {
19        // "/tasks/1": Update a task
20      })
21      .delete((req, res) => {
22        // "/tasks/1": Delete a task
23      });
24  };
```

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.

```
1   // Do you remember this function from auth.js?.
2   Users.findById(payload.id)
3     .then(user => {
4       if (user) {
5         return done(null, {
6           id: user.id,
7           email: user.email
8         });
9       }
10      return done(null, false);
11    })
12    .catch(error => done(error, null));
```

The `done()` callback sends the authenticated user's data to the authenticated routes, which receive these data via the `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 so their queries use as a parameter the req.user.id. To do this, edit routes/tasks.js and, in the route app.route("/tasks"), do the following modifications.

```
1   app.route("/tasks")
2     .all(app.auth.authenticate())
3     .get((req, res) => {
4       Tasks.findAll({
5         where: { user_id: req.user.id }
6       })
7       .then(result => res.json(result))
8       .catch(error => {
9         res.status(412).json({msg: error.message});
10      });
11    })
12    .post((req, res) => {
13      req.body.user_id = req.user.id;
14      Tasks.create(req.body)
15        .then(result => res.json(result))
16        .catch(error => {
17          res.status(412).json({msg: error.message});
18        });
19    });
```

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

```
1   app.route("/tasks/:id")
2     .all(app.auth.authenticate())
3     .get((req, res) => {
4       Tasks.findOne({ where: {
5         id: req.params.id,
6         user_id: req.user.id
7       }})
8       .then(result => {
9         if (result) {
10          return res.json(result);
11        }
12        return res.sendStatus(404);
13      })
14      .catch(error => {
15        res.status(412).json({msg: error.message});
16      });
17    })
18    .put((req, res) => {
19      Tasks.update(req.body, { where: {
20        id: req.params.id,
```

```
21            user_id: req.user.id
22          }})
23          .then(result => res.sendStatus(204))
24          .catch(error => {
25            res.status(412).json({msg: error.message});
26          });
27        })
28        .delete((req, res) => {
29          Tasks.destroy({ where: {
30            id: req.params.id,
31            user_id: req.user.id
32          }})
33          .then(result => res.sendStatus(204))
34          .catch(error => {
35            res.status(412).json({msg: error.message});
36          });
37        });
```

To finish this refactoring, let's modify the users' resources. Now we need to adapt some pieces of code inside the users' routes. Basically, we are going to change the way of searching or deleting a user to use the authenticated user's id.

In this case, it 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 the 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 userid will be used.

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

```
1   module.exports = app => {
2     const Users = app.db.models.Users;
3
4     app.route("/user")
5       .all(app.auth.authenticate())
6       .get((req, res) => {
7         Users.findById(req.user.id, {
8           attributes: ["id", "name", "email"]
9         })
10        .then(result => res.json(result))
11        .catch(error => {
12          res.status(412).json({msg: error.message});
13        });
14      })
15      .delete((req, res) => {
16        Users.destroy({where: {id: req.user.id} })
17          .then(result => res.sendStatus(204))
18          .catch(error => {
```

```
19              res.status(412).json({msg: error.message});
20          });
21      });
22
23  app.post("/users", (req, res) => {
24    Users.create(req.body)
25    .then(result => res.json(result))
26    .catch(error => {
27      res.status(412).json({msg: error.message});
28    });
29  });
30  };
```

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, we now have a safe mechanism for user authentication between client and server.

To this point, we have implemented all of the application back end, and we have not created a client application that uses the power of our API. Don't worry, though: There are several surprises in the next chapters that are going to get you excited. Just keep reading!

CHAPTER 8

■ ■ ■

Testing the Application: Part 1

Creating automated tests is highly recommended. There are several types of tests: unitary, functional, acceptance, and others. This chapter focuses only on the acceptance test, which 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 Mocha (Figure 8-1), which is very popular in the Node.js community.
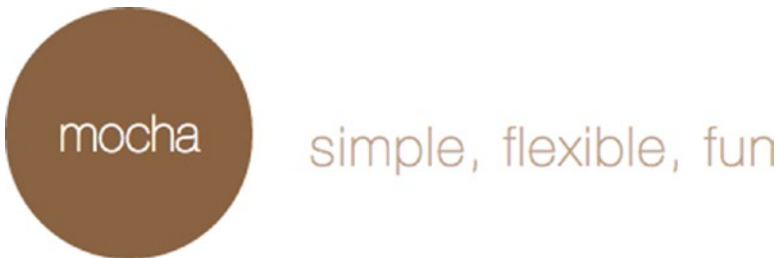
Figure 8-1. *Mocha test runner*

Mocha has the following features:

- TDD style.

- BDD style.

- Code coverage HTML report.

- Customized test reports.

- Asynchronous test support.

- Easy integration with the `should,` `assert`, and `chai` modules.

It is a complete environment for writing tests, and you can learn more by accessing the Mocha web site at `https://mochajs.org`.

Setting Up the Test Environment

To set up our test environment, first we are going to set up a new database that we can use to work with some fake data. This practice is often used to make sure that an application can work easily in multiple environments. For now, our API has just a single environment, because all the examples were developed in the `development` environment.

To enable support for multiple environments, let's rename the current `libs/config.js` file `libs/config.development.js` and then create the `libs/config.test.js` file. The only new parameter in this new file is `logging: false`, which disables the SQL log outputs. It is necessary to disable these logs to avoid difficulty with the test report in the terminal. This file (`libs/config.test.js`) should look like the following code.

```
1   module.exports = {
2     database: "ntask_test",
3     username: "",
4     password: "",
5     params: {
6       dialect: "sqlite",
7       storage: "ntask.sqlite",
8       logging: false,
9       define: {
10        underscored: true
11      }
12    },
13    jwtSecret: "NTASK_TEST",
14    jwtSession: {session: false}
15  };
```

Next we have to set up some files, each one with data specific to its corresponding environments. To load the settings according to the current environment, we must create a code to identify what environment it is. In this case, we can use the `process.env` object, which basically returns 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 this value as a base. Our application will have to load the settings from a `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 re-create the `libs/config.js` file to load the settings according to the right system environment.

```
1   module.exports = app => {
2     const env = process.env.NODE_ENV;
3     if (env) {
4       return require(`./config.${env}.js`);
5     }
6     return require("./config.development.js");
7   };
```

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 execute some requests in the API.

All of these modules will be installed as devDependencies in the package.json file to use them only as test dependencies. To do this, you need to use the --save-dev flag with the npm install command, as shown here.

```
1   npm install babel-register@6.5.2 mocha@2.4.5 chai@3.5.0 supertest@1.2.0
--save-d\
2   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 package.json and include the scripts.test attribute.

```
1   {
2     "name": "ntask-api",
3     "version": "1.0.0",
4     "description": "Task list API",
5     "main": "index.js",
6     "scripts": {
7       "start": "babel-node index.js",
8       "test": "NODE_ENV=test mocha test/**/*.js"
9     },
10    "author": "Caio Ribeiro Pereira",
11    "dependencies": {
12      "babel-cli": "^6.5.1",
13      "babel-preset-es2015": "^6.5.0",
14      "bcrypt": "^0.8.5",
15      "body-parser": "^1.15.0",
16      "consign": "^0.1.2",
17      "express": "^4.13.4",
18      "jwt-simple": "^0.4.1",
19      "passport": "^0.3.2",
20      "passport-jwt": "^2.0.0",
21      "sequelize": "^3.19.2",
22      "sqlite3": "^3.1.1"
23    },
24    "devDependencies": {
25      "babel-register": "^6.5.2",
26      "chai": "^3.5.0",
27      "mocha": "^2.4.5",
28      "supertest": "^1.2.0"
29    }
30  }
```

Then, we are going to export our main API module, `index.js`, to allow the API to be started during the tests. To do so, you must include the `module.exports = app` at 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 test report.

```
1    import express from "express";
2    import consign from "consign";
3
4    const app = express();
5
6    consign({verbose: false})
7      .include("libs/config.js")
8      .then("db.js")
9      .then("auth.js")
10     .then("libs/middlewares.js")
11     .then("routes")
12     .then("libs/boot.js")
13     .into(app);
14
15   module.exports = app;
```

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

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

```
1    module.exports = app => {
2      if (process.env.NODE_ENV !== "test") {
3        app.db.sequelize.sync().done(() => {
4          app.listen(app.get("port"), () => {
5            console.log(`NTask API - Port ${app.get("port")}`);
6          });
7        });
8      }
9    };
```

To finish our test 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 have all the main things load once, we will save milliseconds in test execution. To implement this simple practice, create the file `test/helpers.js`.

```
1    import supertest from "supertest";
2    import chai from "chai";
3    import app from "../index.js";
4
5    global.app = app;
```

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

Then, let's create a simple file that allows us to include some settings as parameters to the mocha command. This will be responsible for loading 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 for Mocha be able to run the tests in ECMAScript 6 standard via the babel-register module.

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

```
1    --require test/helpers
2    --reporter spec
3    --compilers js:babel-register
4    --slow 5000=
```

Writing the First Test

Now that we have finished the setup of the test environment, it is time to test something. We can write some test code for routes/index.js because it is very simple to test: Basically, we can 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 match.

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 the expect(res.body).to.eql(expected) function.

To implement this test, create the test/routes/index.js file using the following code.

```
1    describe("Routes: Index", () => {
2      describe("GET /", () => {
3        it("returns the API status", done => {
4          request.get("/")
5            .expect(200)
6            .end((err, res) => {
7              const expected = {status: "NTask API"};
8              expect(res.body).to.eql(expected);
9              done(err);
10           });
11       });
12     });
13   });
```
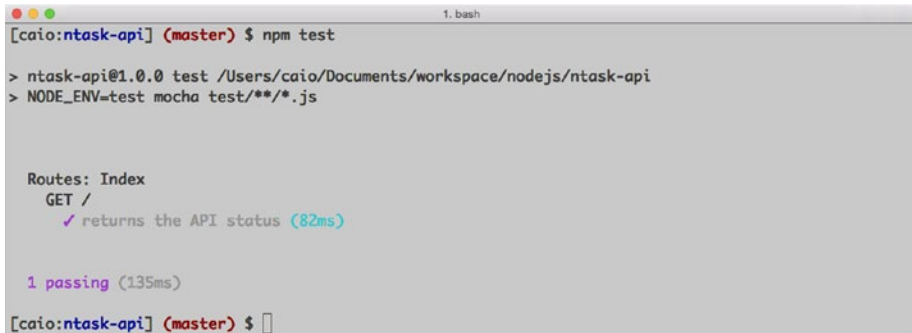
To execute this test, run the following command.

```
1    npm test
```

After the execution, you should have output similar to what is shown in Figure 8-2.

Figure 8-2. *Running the first test*

Testing the Authentication Endpoint

In this section, we implement several tests. To start, let's test the endpoint from routes/
token.js, which is responsible to generate 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 the wrong password.

- Request with an unregistered e-mail.

- Request without an e-mail and password.

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

```
1   describe("Routes: Token", () => {
2     const Users = app.db.models.Users;
3     describe("POST /token", () => {
4       beforeEach(done => {
5         // Runs before each test...
6       });
7       describe("status 200", () => {
8         it("returns authenticated user token", done =>{
9           // Test's logic...
10        });
11      });
12      describe("status 401", () => {
13        it("throws error when password is incorrect", done => {
14          // Test's logic...
15        });
16        it("throws error when email not exist", done => {
17          // Test's logic...
18        });
19         it("throws error when email and password are blank", done => {
```

```
20          // Test's logic...
21        });
22      });
23    });
24  });
```

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

```
1   beforeEach(done => {
2     Users
3       .destroy({where: {}})
4       .then(() => Users.create({
5         name: "John",
6         email: "john@mail.net",
7         password: "12345"
8       }))
9       .then(() => done());
10  });
```

Now, we are going to implement things 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 e-mail and password of a valid user via the send() function.

To finish the test, the end(err, res) callback must return res.body with the token key to be checked via the 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 the done(err) function, because if something goes wrong during the test, this function will show the details of the error. Here is the complete code of this first test.

```
1   it("returns authenticated user token", done => {
2     request.post("/token")
3       .send({
4         email: "john@mail.net",
5         password: "12345"
6       })
7       .expect(200)
8       .end((err, res) => {
9         expect(res.body).to.include.keys("token");
10        done(err);
11      });
12  });
```

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

```
1   it("throws error when password is incorrect", done => {
2     request.post("/token")
3       .send({
4         email: "john@mail.net",
5         password: "WRONG_PASSWORD"
6       })
7       .expect(401)
8       .end((err, res) => {
9         done(err);
10      });
11  });
```

The next test is very similar to the last one, but now, it tests an invalid user's e-mail, expecting the request to return the 401 status code again.

```
1   it("throws error when email not exist", done => {
2     request.post("/token")
3       .send({
4         email: "wrong@email.com",
5         password: "12345"
6       })
7       .expect(401)
8       .end((err, res) => {
9         done(err);
10      });
11  });
```

To finish this test case, let's check if we get the same 401 status code when no e-mail and no password are sent. This one is even simpler, because we don't need to send parameters in this request.
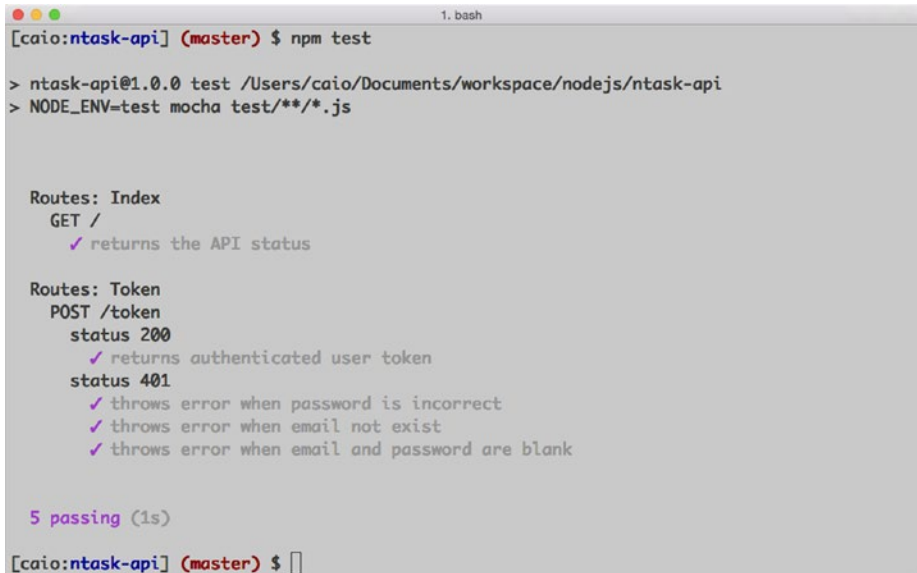
```
1   it("throws error when email and password are blank", done => {
2     request.post("/token")
3       .expect(401)
4       .end((err, res) => {
5         done(err);
6       });
7   });
```

Conclusion

To run all tests now, just run the `npm test` command again. You should see a result similar to Figure 8-3.

Figure 8-3. *Test result*

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

■ ■ ■

Testing the Application: Part 2

Continuing the test implementation, now focus next on writing some tests for the resources: tasks and users.

Testing a Task's Endpoints

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

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

```
1   import jwt from "jwt-simple";
2
3   describe("Routes: Tasks", () => {
4     const Users = app.db.models.Users;
5     const Tasks = app.db.models.Tasks;
6     const jwtSecret = app.libs.config.jwtSecret;
7     let token;
8     let fakeTask;
9     beforeEach(done => {
10      // Runs before each test...
11    });
12    describe("GET /tasks", () => {
13      describe("status 200", () => {
14        it("returns a list of tasks", done => {
15          // Test's logic...
16        });
17      });
18    });
19    describe("POST /tasks/", () => {
20      describe("status 200", () => {
21        it("creates a new task", done => {
```

```
22              // Test's logic...
23           });
24         });
25       });
26       describe("GET /tasks/:id", () => {
27         describe("status 200", () => {
28           it("returns one task", done => {
29             // Test's logic...
30           });
31         });
32         describe("status 404", () => {
33           it("throws error when task not exist", done => {
34             // Test's logic...
35           });
36         });
37       });
38       describe("PUT /tasks/:id", () => {
39         describe("status 204", () => {
40           it("updates a task", done => {
41             // Test's logic...
42           });
43         });
44       });
45       describe("DELETE /tasks/:id", () => {
46         describe("status 204", () => {
47           it("removes a task", done => {
48             // Test's logic...
49           })
50         });
51       });
52     });
```

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

We'll do the same with task creation, but instead of using the function Tasks.create() it will be 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.

```
1   beforeEach(done => {
2     Users
3       .destroy({where: {}})
4       .then(() => Users.create({
5         name: "John",
6         email: "john@mail.net",
7         password: "12345"
8       }))
9       .then(user => {
10        Tasks
11          .destroy({where: {}})
12          .then(() => Tasks.bulkCreate([{
13            id: 1,
14            title: "Work",
15            user_id: user.id
16          }, {
17            id: 2,
18            title: "Study",
19            user_id: user.id
20          }]))
21          .then(tasks => {
22            fakeTask = tasks[0];
23            token = jwt.encode({id: user.id}, jwtSecret);
24            done();
25          });
26      });
27    })
```

With the pretest routine ready, we are going to write all the tasks tests, starting with the GET / route. Performed a request on it via the request.get("/tasks") function, also using the function set("Authorization", JWT ${token}) to allow sending a header on the request. In this case, the header Authorization is sent along with the value of a valid token.

To make sure the test is successfully accomplished, follow these steps.

1. Check the status 200 via the expect(200) function.

2. Apply a simple validation to ensure that the array of size 2 will be returned via the expect(res.body).to.have.length 2) function.

3. Compare if the titles of the first two tasks are the same as those created by the function Tasks.bulkCreate().

```
1   describe("GET /tasks", () => {
2     describe("status 200", () => {
3       it("returns a list of tasks", done => {
4         request.get("/tasks")
5           .set("Authorization", `JWT ${token}`)
6           .expect(200)
```

```
7           .end((err, res) => {
8             expect(res.body).to.have.length(2);
9             expect(res.body[0].title).to.eql("Work");
10            expect(res.body[1].title).to.eql("Study");
11            done(err);
12          });
13        });
14      });
15    })
```

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

```
1    describe("POST /tasks", () => {
2      describe("status 200", () => {
3        it("creates a new task", done => {
4          request.post("/tasks")
5            .set("Authorization", `JWT ${token}`)
6            .send({title: "Run"})
7            .expect(200)
8            .end((err, res) => {
9              expect(res.body.title).to.eql("Run");
10             expect(res.body.done).to.be.false;
11             done(err);
12           });
13        });
14      });
15    });
```

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 an id of an invalid task is used, we are going to use the function expect(404) to test the status 404 that indicates if the request did not find a resource.

```
1    describe("GET /tasks/:id", () => {
2      describe("status 200", () => {
3        it("returns one task", done => {
4          request.get(`/tasks/${fakeTask.id}`)
5            .set("Authorization", `JWT ${token}`)
6            .expect(200)
7            .end((err, res) => {
8              expect(res.body.title).to.eql("Work");
9              done(err);
10           });
11        });
```

```
12      });
13      describe("status 404", () => {
14        it("throws error when task not exist", done => {
15          request.get("/tasks/0")
16            .set("Authorization", `JWT ${token}`)
17            .expect(404)
18            .end((err, res) => done(err));
19        });
20      });
21    }
```
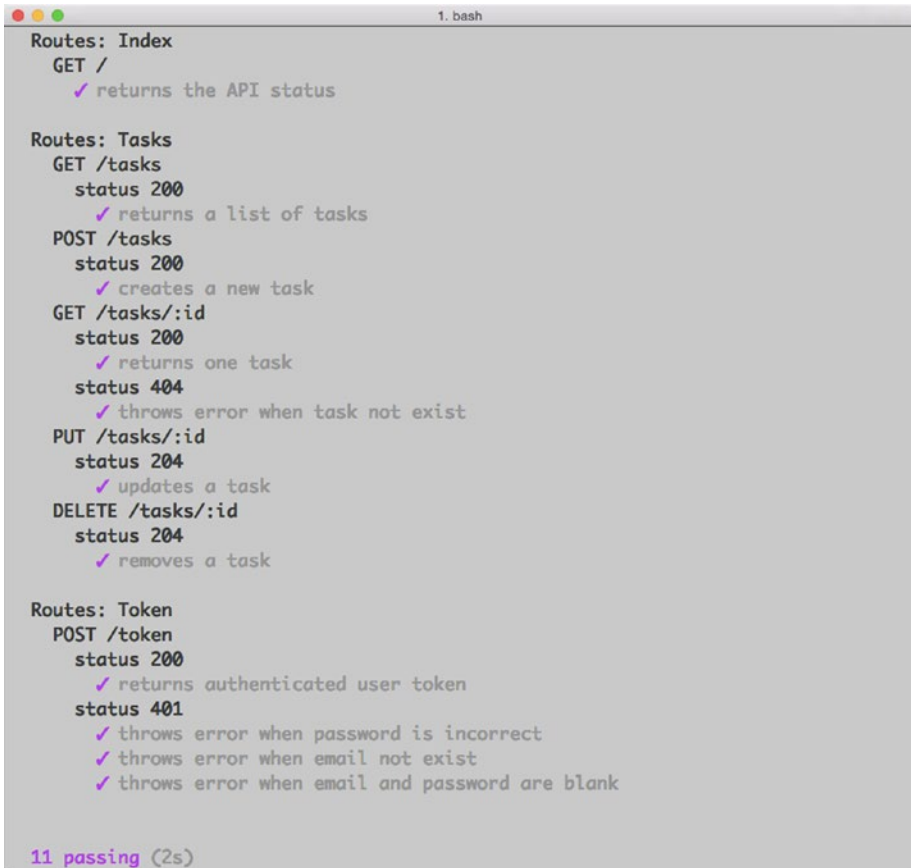
To finish the tests, we are going to test the successful behavior of the routes PUT /tasks/:id and DELETE /tasks/:id. Both of them will use the same functions, except that one test is going to execute the function request.put() and the other one is going to execute the function request.delete(). Both of them, though, expect that the request returns a 204 status code via the expect(204) function.

```
1     describe("PUT /tasks/:id", () => {
2       describe("status 204", () => {
3         it("updates a task", done => {
4           request.put(`/tasks/${fakeTask.id}`)
5             .set("Authorization", `JWT ${token}`)
6             .send({
7               title: "Travel",
8               done: true
9             })
10            .expect(204)
11            .end((err, res) => done(err));
12        });
13      });
14    });
15    describe("DELETE /tasks/:id", () => {
16      describe("status 204", () => {
17        it("removes a task", done => {
18          request.delete(`/tasks/${fakeTask.id}`)
19            .set("Authorization", `JWT ${token}`)
20            .expect(204)
21            .end((err, res) => done(err));
22        });
23      });
24    })
```

We have finished the tests of tasks resources. If you execute the command npm test again, you will see the result shown in Figure 9-1.

Figure 9-1. *Testing a task's resources*

Testing a User's Endpoints

To test the user's resource is even simpler, because basically we are going what was explained in the last tests. To start, create the file test/routes/users.js with the following structure.

```
1    import jwt from "jwt-simple";
2
3    describe("Routes: Tasks", () => {
4      const Users = app.db.models.Users;
5      const jwtSecret = app.libs.config.jwtSecret;
6      let token;
7      beforeEach(done => {
8        // Runs before each test...
9      });
```

```
10    describe("GET /user", () => {
11      describe("status 200", () => {
12        it("returns an authenticated user", done => {
13          // Test's logic...
14        });
15      });
16    });
17    describe("DELETE /user", () => {
18      describe("status 204", () => {
19        it("deletes an authenticated user", done => {
20          // Test's logic...
21        });
22      });
23    });
24    describe("POST /users", () => {
25      describe("status 200", () => {
26        it("creates a new user", done => {
27          // Test's logic...
28        });
29      });
30    });
31  });
```

The pretest logic is going to be simplified, but it will have the generation of a valid token. This code shows how to implement the beforeEach(done) function.

```
1   beforeEach(done => {
2     Users
3       .destroy({where: {}})
4       .then(() => Users.create({
5         name: "John",
6         email: "john@mail.net",
7         password: "12345"
8       }))
9       .then(user => {
10        token = jwt.encode({id: user.id}, jwtSecret);
11        done();
12      });
13  });
```

Now, to implement these tests, let's get started testing the GET /user route, which must return the authenticated user's data, which basically sends a token and receives as a response the user's data created by the beforeEach(done) function.

```
1   describe("GET /user", () => {
2     describe("status 200", () => {
3       it("returns an authenticated user", done => {
4         request.get("/user")
```

```
5              .set("Authorization", `JWT ${token}`)
6              .expect(200)
7              .end((err, res) => {
8                expect(res.body.name).to.eql("John");
9                expect(res.body.email).to.eql("john@mail.net");
10               done(err);
11           });
12         });
13       });
14     })
```

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

```
1   describe("DELETE /user", () => {
2     describe("status 204", () => {
3       it("deletes an authenticated user", done => {
4         request.delete("/user")
5           .set("Authorization", `JWT ${token}`)
6           .expect(204)
7           .end((err, res) => done(err));
8       });
9     });
10  });
```
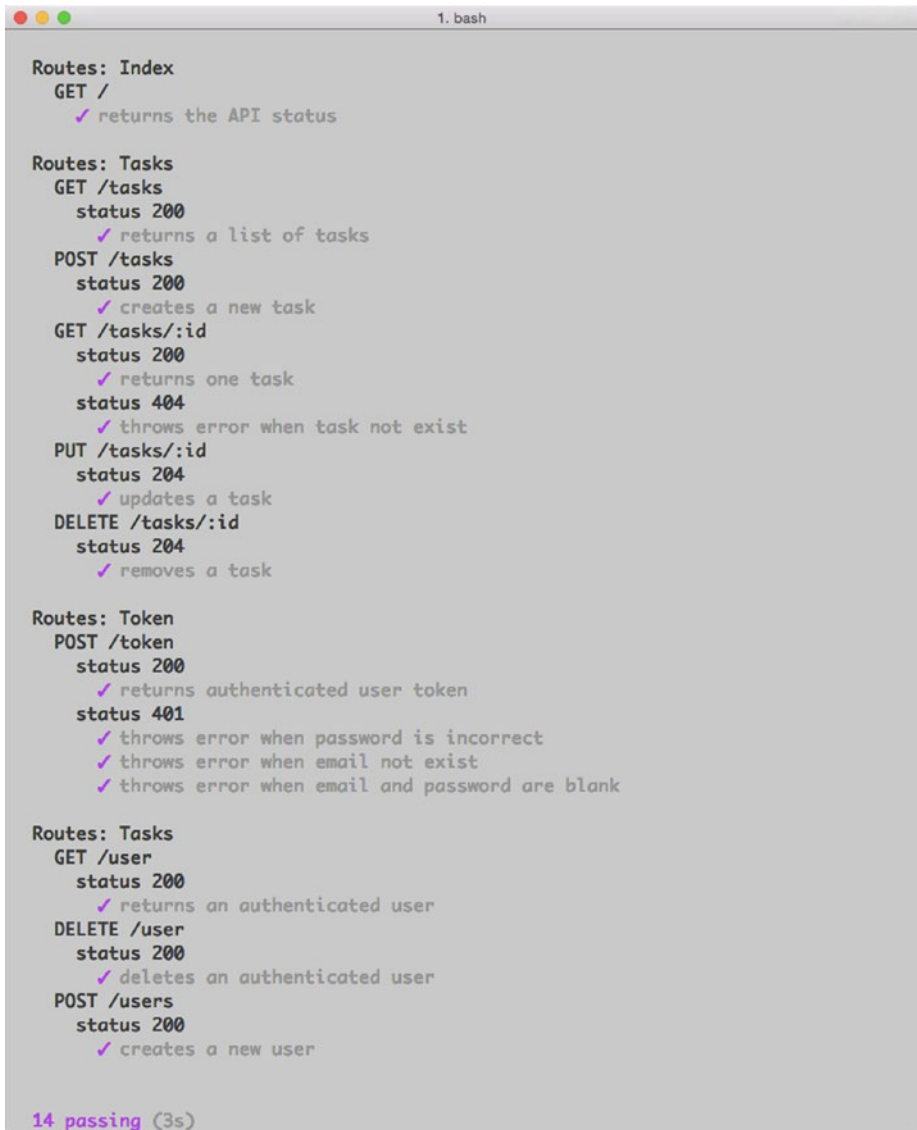
To finish the last test, we are going to implement the test for new user's route. This one, doesn't require a token; after all, it is an open route for new users to register an account in the API. This test's code is shown here.

```
1   describe("POST /users", () => {
2     describe("status 200", () => {
3       it("creates a new user", done => {
4         request.post("/users")
5           .send({
6             name: "Mary",
7             email: "mary@mail.net",
8             password: "12345"
9           })
10          .expect(200)
11          .end((err, res) => {
12            expect(res.body.name).to.eql("Mary");
13            expect(res.body.email).to.eql("mary@mail.net");
14            done(err);
15          });
16       });
17     });
18  });
```

Now, if you execute the command `npm test` again, you'll see a report like the one shown in Figure 9-2.

Figure 9-2. *Testing user's resources*

Conclusion

If you have reached this step, then you have developed a small but powerful API using Node.js and SQL database. Everything is already working and has been tested to ensure the quality of the project code. In the next chapter, we explore a very useful tool for generating API documentation.

CHAPTER 10

■ ■ ■

Documenting the API

If you have reached this chapter and your application is working correctly—with routes to the management tasks and users, integrated with a database, and with user authentication via JSON Web Token—congratulations! You have created, following some best practices, a 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 enough of an application to deploy it into a production environment.

Introduction to ApiDoc.js

In this chapter, we'll learn how to write and generate 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 best part is that we are going to use a very simple tool and all the documentation of our application will be built using the code's comments.

Our project will use ApiDoc.js (see Figure 10-1), which is a Node.js module to generate elegant documentation for APIs.
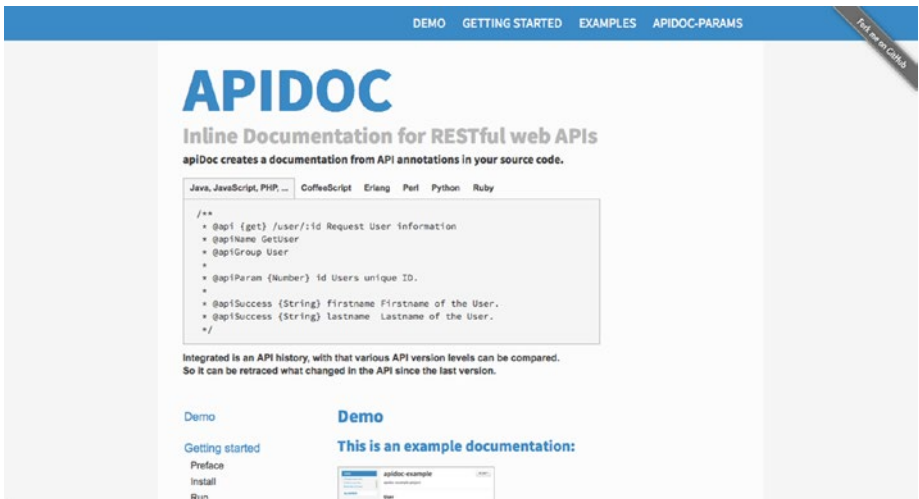
Figure 10-1. *ApiDoc.js home page*

This module is a command-line interface (CLI) 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 an `npm alias` command to use every time we start the API server. Therefore, its installation is going to be as a local module, similar to the others. Install it using this command.

```
1   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 starting the API server. We also include the attribute `apidoc.name` to set a title and `apidoc.template.forceLanguage` to set the default language to be used in the API documentation.

Open and edit the `package.json`, making these changes.

```
1   {
2     "name": "ntask-api",
3     "version": "1.0.0",
4     "description": "Task list API",
5     "main": "index.js",
6     "scripts": {
7       "start": "npm run apidoc && babel-node index.js",
8       "apidoc": "apidoc -i routes/ -o public/apidoc",
9       "test": "NODE_ENV=test mocha test/**/*.js"
10    },
11    "apidoc": {
12      "name": "Node Task API - Documentation",
13      "template": {
14        "forceLanguage": "en"
15      }
16    },
17    "author": "Caio Ribeiro Pereira",
18    "dependencies": {
19      "babel-cli": "^6.5.1",
20      "babel-preset-es2015": "^6.5.0",
21      "bcrypt": "^0.8.5",
22      "body-parser": "^1.15.0",
23      "consign": "^0.1.2",
24      "express": "^4.13.4",
25      "jwt-simple": "^0.4.1",
26      "passport": "^0.3.2",
27      "passport-jwt": "^2.0.0",
28      "sequelize": "^3.19.2",
29      "sqlite3": "^3.1.1"
30    },
31    "devDependencies": {
32      "apidoc": "^0.15.1",
33      "babel-register": "^6.5.2",
```

```
34       "chai": "^3.5.0",
35       "mocha": "^2.4.5",
36       "supertest": "^1.2.0"
37    }
38  }
```

Now, every time you run the command npm start, if you want to generate new documentation without initiating the server, you can just run the npm run apidoc command. Both of the commands are going to search all the existing comments in the routes directory to generate 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 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. The code should look like this.

```
1    import bodyParser from "body-parser";
2    import express from "express";
3
4    module.exports = app => {
5      app.set("port", 3000);
6      app.set("json spaces", 4);
7      app.use(bodyParser.json());
8      app.use(app.auth.initialize());
9      app.use((req, res, next) => {
10       delete req.body.id;
11       next();
12     });
13     app.use(express.static("public"));
14   };
```

To validate if everything is working correctly, let's start documenting 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 response.

- @apiSuccessExample: Shows an output sample of a successful response.

To document this endpoint, edit the file routes/index.js using this code.

```
1    module.exports = app => {
2      /**
3       * @api {get} / API Status
```

```
 4      * @apiGroup Status
 5      * @apiSuccess {String} status API Status' message
 6      * @apiSuccessExample {json} Success
 7      * HTTP/1.1 200 OK
 8      * {"status": "NTask API"}
 9      */
10    app.get("/", (req, res) => {
11      res.json({status: "NTask API"});
12    });
13  };
```

To test these changes, restart the server, then open the browser and go to
http://localhost:3000/apidoc. If no errors occur, you will see a beautiful
documentation page about your API, as shown in Figure 10-2.
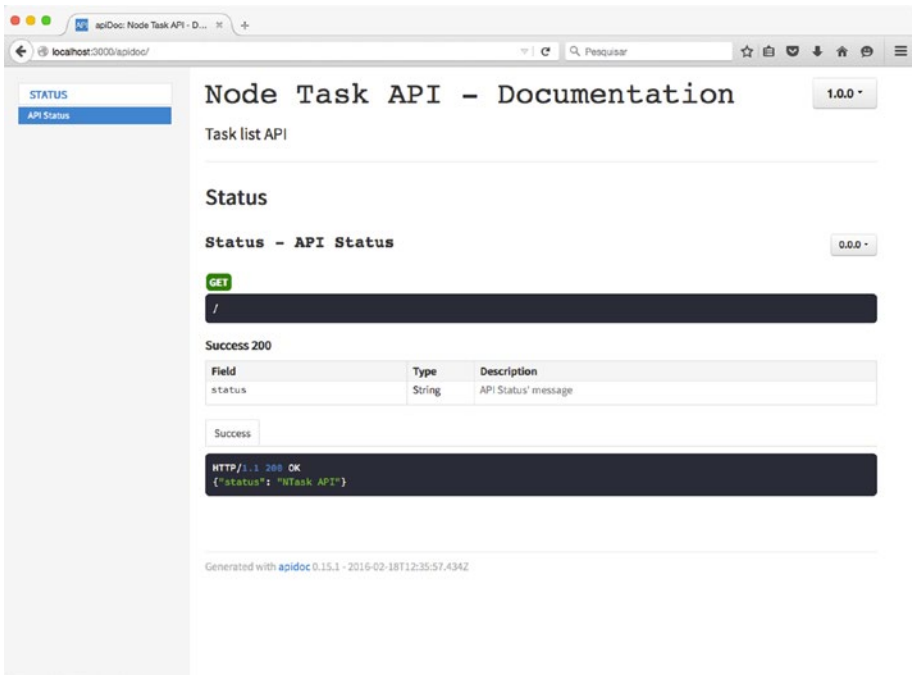
Figure 10-2. *API documentation page*

Documenting Token Generation

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

To start, 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 might or might not be required to be submitted in a request.

- @apiParamExample: Shows a sample of an input parameter; in our case, we will display a JSON input format.

- @apiErrorExample: Shows a sample of some errors that could be generated by the API if something goes wrong.

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

```
1   import jwt from "jwt-simple";
2
3   module.exports = app => {
4     const cfg = app.libs.config;
5     const Users = app.db.models.Users;
6
7     /**
8      * @api {post} /token Authentication Token
9      * @apiGroup Credentials
10     * @apiParam {String} email User email
11     * @apiParam {String} password User password
12     * @apiParamExample {json} Input
13     *    {
14     *      "email": "john@connor.net",
15     *      "password": "123456"
16     *    }
17     * @apiSuccess {String} token Token of authenticated user
18     * @apiSuccessExample {json} Success
19     *    HTTP/1.1 200 OK
20     *    {"token": "xyz.abc.123.hgf"}
21     * @apiErrorExample {json} Authentication error
22     *    HTTP/1.1 401 Unauthorized
23     */
24    app.post("/token", (req, res) => {
25      // The code here was explained in Chapter 7.
26    });
27  };
```

Documenting User Resource

In this section and in the next one, we focus on documenting 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 the header to be used in the request.

Open routes/users.js and let's document it. First, we'll document the GET /user endpoint.

```
1   module.exports = app => {
2   const Users = app.db.models.Users;
3
4   app.route("/user")
5     .all(app.auth.authenticate())
6     /**
7      * @api {get} /user Return the authenticated user's data
8      * @apiGroup User
9      * @apiHeader {String} Authorization Token of authenticated user
10     * @apiHeaderExample {json} Header
11     *    {"Authorization": "JWT xyz.abc.123.hgf"}
12     * @apiSuccess {Number} id User id
13     * @apiSuccess {String} name User name
14     * @apiSuccess {String} email User email
15     * @apiSuccessExample {json} Success
16     *    HTTP/1.1 200 OK
17     *    {
18     *      "id": 1,
19     *      "name": "John Connor",
20     *      "email": "john@connor.net"
21     *    }
22     * @apiErrorExample {json} Find error
23     *    HTTP/1.1 412 Precondition Failed
24     */
25     .get((req, res) => {
26       // GET /user logic...
27     })
```

Then, let's document the route DELETE /user.

```
1   /**
2    * @api {delete} /user Deletes an authenticated user
3    * @apiGroup User
4    * @apiHeader {String} Authorization Token of authenticated user
5    * @apiHeaderExample {json} Header
6    * {"Authorization": "JWT xyz.abc.123.hgf"}
7    * @apiSuccessExample {json} Success
8    * HTTP/1.1 204 No Content
9    * @apiErrorExample {json} Delete error
10   * HTTP/1.1 412 Precondition Failed
```

```
11    */
12    .delete((req, res) => {
13    // DELETE /user logic...
14    })
```

To finish this endpoint, we need to document its last route, the POST /user. Now we'll use several items to describe the input and output fields of this route.

```
1     /**
2      * @api {post} /users Register a new user
3      * @apiGroup User
4      * @apiParam {String} name User name
5      * @apiParam {String} email User email
6      * @apiParam {String} password User password
7      * @apiParamExample {json} Input
8      *    {
9      *      "name": "John Connor",
10     *      "email": "john@connor.net",
11     *      "password": "123456"
12     *    }
13     * @apiSuccess {Number} id User id
14     * @apiSuccess {String} name User name
15     * @apiSuccess {String} email User email
16     * @apiSuccess {String} password User encrypted password
17     * @apiSuccess {Date} updated_at Update's date
18     * @apiSuccess {Date} created_at Register's date
19     * @apiSuccessExample {json} Success
20     *    HTTP/1.1 200 OK
21     *    {
22     *      "id": 1,
23     *      "name": "John Connor",
24     *      "email": "john@connor.net",
25     *      "password": "$2a$10$SK1B1",
26     *      "updated_at": "2016-02-10T15:20:11.700Z",
27     *      "created_at": "2016-02-10T15:29:11.700Z",
28     *    }
29     * @apiErrorExample {json} Register error
30     *    HTTP/1.1 412 Precondition Failed
31     */
32    app.post("/users", (req, res) => {
33      // POST /users logic...
34    });
35    };
```

Documenting Tasks Resource

Continuing our API documentation, now we need to finish the tasks routes documentation. Let's edit the routes/tasks.js file, initially describing the GET /tasks route.

```
1   module.exports = app => {
2     const Tasks = app.db.models.Tasks;
3
4   app.route("/tasks")
5     .all(app.auth.authenticate())
6     /**
7      * @api {get} /tasks List the user's tasks
8      * @apiGroup Tasks
9      * @apiHeader {String} Authorization Token of authenticated user
10     * @apiHeaderExample {json} Header
11     *    {"Authorization": "JWT xyz.abc.123.hgf"}
12     * @apiSuccess {Object[]} tasks Task list
13     * @apiSuccess {Number} tasks.id Task id
14     * @apiSuccess {String} tasks.title Task title
15     * @apiSuccess {Boolean} tasks.done Task is done?
16     * @apiSuccess {Date} tasks.updated_at Update's date
17     * @apiSuccess {Date} tasks.created_at Register's date
18     * @apiSuccess {Number} tasks.user_id Id do usuário
19     * @apiSuccessExample {json} Success
20     *    HTTP/1.1 200 OK
21     *    [{
22     *      "id": 1,
23     *      "title": "Study",
24     *      "done": false
25     *      "updated_at": "2016-02-10T15:46:51.778Z",
26     *      "created_at": "2016-02-10T15:46:51.778Z",
27     *      "user_id": 1
28     *    }]
29     * @apiErrorExample {json} List error
30     *    HTTP/1.1 412 Precondition Failed
31     */
32     .get((req, res) => {
33      // GET /tasks logic...
34     })
```

Then, let's document the route POST /tasks.

```
1    /**
2     * @api {post} /tasks Register a new task
3     * @apiGroup Tasks
4     * @apiHeader {String} Authorization Token of authenticated user
5     * @apiHeaderExample {json} Header
6     *    {"Authorization": "JWT xyz.abc.123.hgf"}
7     * @apiParam {String} title Task title
8     * @apiParamExample {json} Input
9     *    {"title": "Study"}
10    * @apiSuccess {Number} id Task id
11    * @apiSuccess {String} title Task title
```

```
12    * @apiSuccess {Boolean} done=false Task is done?
13    * @apiSuccess {Date} updated_at Update's date
14    * @apiSuccess {Date} created_at Register's date
15    * @apiSuccess {Number} user_id User id
16    * @apiSuccessExample {json} Success
17    *    HTTP/1.1 200 OK
18    *    {
19    *      "id": 1,
20    *      "title": "Study",
21    *      "done": false,
22    *      "updated_at": "2016-02-10T15:46:51.778Z",
23    *      "created_at": "2016-02-10T15:46:51.778Z",
24    *      "user_id": 1
25    *    }
26    * @apiErrorExample {json} Register error
27    *    HTTP/1.1 412 Precondition Failed
28    */
29    .post((req, res) => {
30    // POST /tasks logic...
31    })
```

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

```
1     /**
2      * @api {get} /tasks/:id Get a task
3      * @apiGroup Tasks
4      * @apiHeader {String} Authorization Token of authenticated user
5      * @apiHeaderExample {json} Header
6      *    {"Authorization": "JWT xyz.abc.123.hgf"}
7      * @apiParam {id} id Task id
8      * @apiSuccess {Number} id Task id
9      * @apiSuccess {String} title Task title
10     * @apiSuccess {Boolean} done Task is done?
11     * @apiSuccess {Date} updated_at Update's date
12     * @apiSuccess {Date} created_at Register's date
13     * @apiSuccess {Number} user_id User id
14     * @apiSuccessExample {json} Success
15     *    HTTP/1.1 200 OK
16     *    {
17     *      "id": 1,
18     *      "title": "Study",
19     *      "done": false
20     *      "updated_at": "2016-02-10T15:46:51.778Z",
21     *      "created_at": "2016-02-10T15:46:51.778Z",
22     *      "user_id": 1
23     *    }
24     * @apiErrorExample {json} Task not found error
```

```
25      *     HTTP/1.1 404 Not Found
26      * @apiErrorExample {json} Find error
27      *     HTTP/1.1 412 Precondition Failed
28      */
29      .get((req, res) => {
30       // GET /tasks/:id logic...
31      })
```

Next is PUT /tasks/:id.

```
1       /**
2        * @api {put} /tasks/:id Update a task
3        * @apiGroup Tasks
4        * @apiHeader {String} Authorization Token of authenticated user
5        * @apiHeaderExample {json} Header
6        *     {"Authorization": "JWT xyz.abc.123.hgf"}
7        * @apiParam {id} id Task id
8        * @apiParam {String} title Task title
9        * @apiParam {Boolean} done Task is done?
10       * @apiParamExample {json} Input
11       *     {
12       *        "title": "Work",
13       *        "done": true
14       *     }
15       * @apiSuccessExample {json} Success
16       *     HTTP/1.1 204 No Content
17       * @apiErrorExample {json} Update error
18       *     HTTP/1.1 412 Precondition Failed
19       */
20      .put((req, res) => {
21       // PUT /tasks/:id logic...
22      })
```

Finally, let's finish this chapter by documenting the route DELETE /tasks/:id.

```
1       /**
2        * @api {delete} /tasks/:id Remove a task
3        * @apiGroup Tasks
4        * @apiHeader {String} Authorization Token of authenticated user
5        * @apiHeaderExample {json} Header
6        *     {"Authorization": "JWT xyz.abc.123.hgf"}
7        * @apiParam {id} id Task id
8        * @apiSuccessExample {json} Success
9        *     HTTP/1.1 204 No Content
10       * @apiErrorExample {json} Delete error
11       * HTTP/1.1 412 Precondition Failed
12       */
13      .delete((req, res) => {
```

```
14        // DELETE /tasks/:id logic...
15      });
16    };
```

Let's test our efforts. Just restart the server and then go to http://localhost:3000/apidoc.

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, as shown in Figure 10-3.
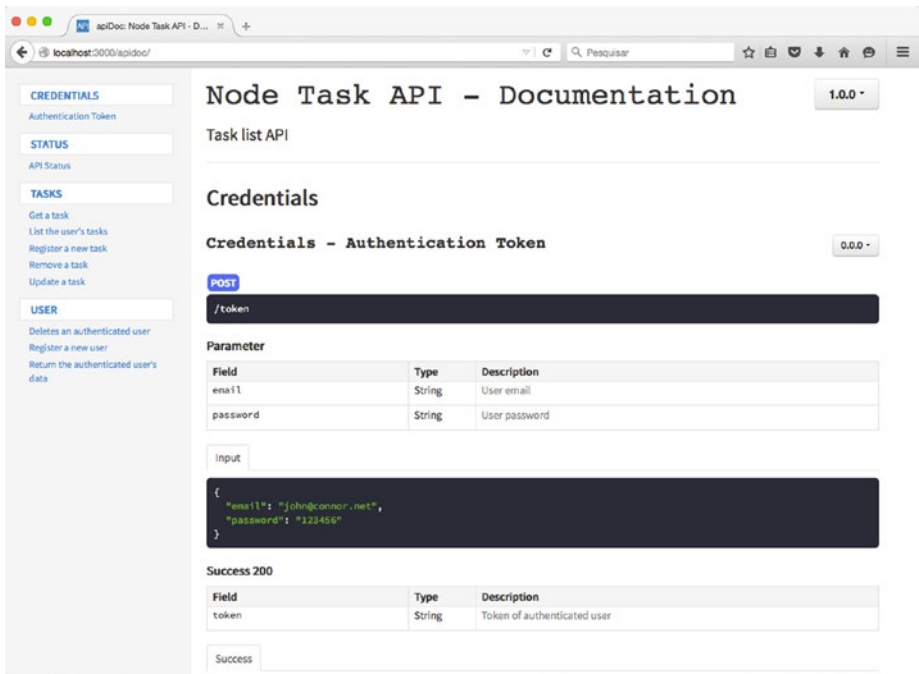
Figure 10-3. *Now our API is well documented*

Conclusion

Congratulations! You have finished another excellent chapter. Now, in addition to a functional API, you also have complete documentation to allow other developers to create client-side applications using your API.

Keep reading, because in the next chapter, we include some frameworks and best practices for our API so that it works fine in a production environment.

91

CHAPTER 11

■ ■ ■

Preparing the Production Environment

Cross-origin resource sharing (CORS) is an important HTTP mechanism. It is responsible for allowing or denying asynchronous requests from other domains.

CORS, in practice, includes only the HTTP headers that are included on the server side. Those headers can indicate which domain can consume the API, which HTTP methods are allowed, and, mainly, which endpoints can be shared publicly for applications from other domains to consume. Generally when implementing CORS in a browser, each cross-origin request will be preceded by an OPTIONS request that checks whether the GET or POST will be okay.

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 to become public, meaning that any client can make requests on our API. For security reasons, then, let's install and use the module cors to limit which resources a client can gain access to from another domain.

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

To initiate it, just use the middleware app.use(cors()) in libs/middlewares.js.

```
1    import bodyParser from "body-parser";
2    import express from "express";
3    import cors from "cors";
4
5    module.exports = app => {
6      app.set("port", 3000);
7      app.set("json spaces", 4);
8      app.use(cors());
9      app.use(bodyParser.json());
10     app.use(app.auth.initialize());
```

```
11     app.use((req, res, next) => {
12       delete req.body.id;
13       next();
14     });
15     app.use(express.static("public"));
16 };
```

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 to, which methods they can use, and, mainly, which headers must be required for the clients to include in the request. In our case, let's set up only three attributes: origin (allowed domains), methods (allowed methods), and allowedHeaders (requested headers).

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

```
1    import bodyParser from "body-parser";
2    import express from "express";
3    import cors from "cors";
4
5    module.exports = app => {
6      app.set("port", 3000);
7      app.set("json spaces", 4);
8      app.use(cors({
9        origin: ["http://localhost:3001"],
10       methods: ["GET", "POST", "PUT", "DELETE"],
11       allowedHeaders: ["Content-Type", "Authorization"]
12     }));
13     app.use(bodyParser.json());
14     app.use(app.auth.initialize());
15     app.use((req, res, next) => {
16       delete req.body.id;
17       next();
18     });
19     app.use(express.static("public"));
20 }
```

Now we have an API that is only going to allow client apps from the address http://localhost:3001. Don't be concerned, because this will be the domain of the client application that we'll build in the next chapters.

A Bit More About CORS

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

Generating Logs

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

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

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

After that, let's write the code responsible for setting up and loading the winston module. We are going to verify if the logs folder exists, using the native module fs (file system) to treat files.

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

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

```
1   import fs from "fs";
2   import winston from "winston";
3
4   if (!fs.existsSync("logs")) {
5     fs.mkdirSync("logs");
6   }
7
8   module.exports = new winston.Logger({
9     transports: [
10      new winston.transports.File({
11        level: "info",
12        filename: "logs/app.log",
13        maxsize: 1048576,
14        maxFiles: 10,
15        colorize: false
16      })
17    ]
18  });
```

Now, we are going to use our libs/logger.js for two important aspects of our application. The first is to generate SQL log 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 the logger.info() function. To do this, just edit libs/config.development.js as shown here.

```
1   import logger from "./logger.js";
2
3   module.exports = {
4     database: "ntask",
5     username: "",
6     password: "",
7     params: {
8       dialect: "sqlite",
9       storage: "ntask.sqlite",
10      logging: (sql) => {
11        logger.info(`[${new Date()}] ${sql}`);
12      },
13      define: {
14        underscored: true
15      }
16    },
17    jwtSecret: "Nta$K-AP1",
18    jwtSession: {session: false}
19  };
```

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 at 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, `logger.info(message)`. To understand this implementation better, edit the file `libs/middlewares.js` as the following code shows.

```
1   import bodyParser from "body-parser";
2   import express from "express";
3   import morgan from "morgan";
4   import cors from "cors";
5   import logger from "./logger.js";
6
7   module.exports = app => {
8     app.set("port", 3000);
9     app.set("json spaces", 4);
10    app.use(morgan("common", {
11      stream: {
12        write: (message) => {
13          logger.info(message);
14        }
15      }
16    }));
17    app.use(cors({
18      origin: ["http://localhost:3001"],
19      methods: ["GET", "POST", "PUT", "DELETE"],
20      allowedHeaders: ["Content-Type", "Authorization"]
```

```
21     }));
22     app.use(bodyParser.json());
23     app.use(app.auth.initialize());
24     app.use((req, res, next) => {
25       delete req.body.id;
26       next();
27     });
28     app.use(express.static("public"));
29   };
```

To test the log generation, restart the server and access many times any API address, such as http://localhost:3000. After making some requests, access the logs folder. There will be a file containing the logs of the requests similar to the one shown in Figure 11-1.
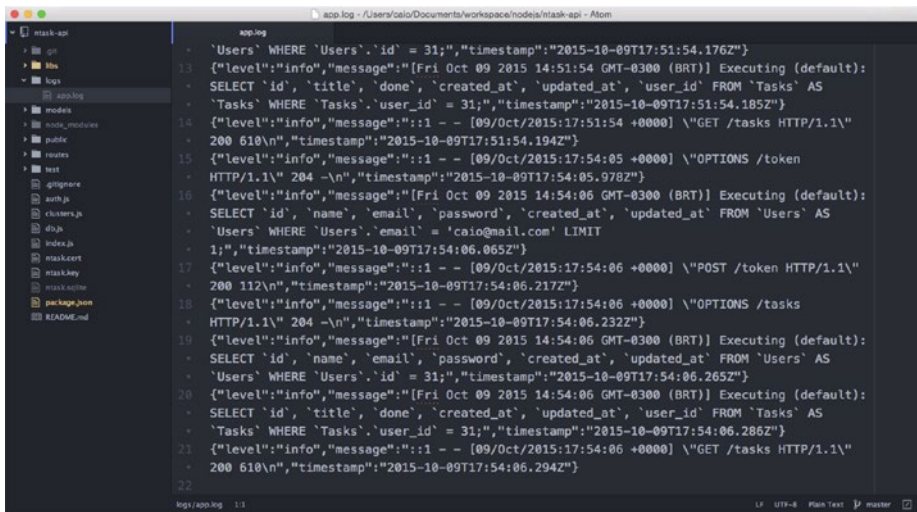
Figure 11-1. *Log of the requests*

Configuring Parallel Processing Using Cluster Module

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

This module 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 active clusters. You determine the number of processes to be created, but a good practice is to instantiate a number of processes based on the amount of server processor cores, or

also a relative amount to core × processors. For example, if you have a single processor of eight cores, you can instantiate eight processes, creating a network of eight clusters. However, if there are four processors of eight cores each, you can create a network of 32 clusters in action.

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

Another advantage is that the clusters are independent. That is, if one 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 successful return of the cluster that went down.

Basing our work on these concepts, we are going to apply in practice the implementation of clusters. Create in the root directory the file clusters.js using the following code.
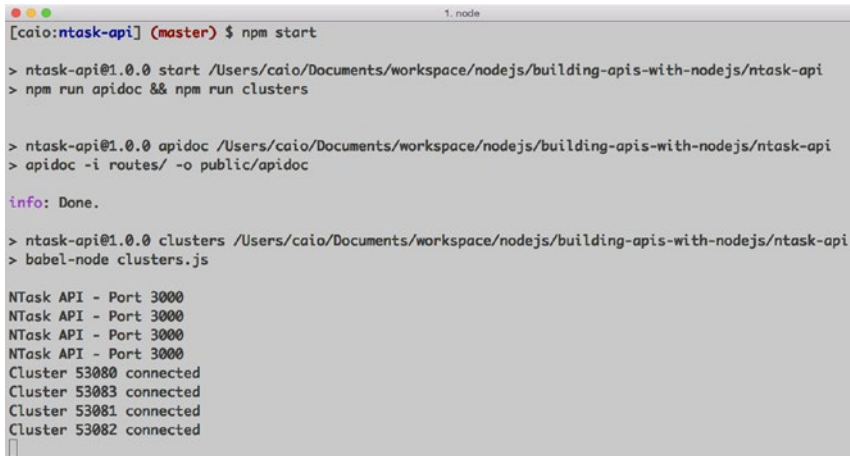
```
1   import cluster from "cluster";
2   import os from "os";
3
4   const CPUS = os.cpus();
5   if (cluster.isMaster) {
6     CPUS.forEach(() => cluster.fork());
7     cluster.on("listening", worker => {
8       console.log("Cluster %d connected", worker.process.pid);
9     });
10    cluster.on("disconnect", worker => {
11      console.log("Cluster %d disconnected", worker.process.pid);
12    });
13    cluster.on("exit", worker => {
14      console.log("Cluster %d is dead", worker.process.pid);
15      cluster.fork();
16      // Ensure starts of a new cluster if an old one dies
17    });
18  } else {
19    require("./index.js");
20 }
```

This time, to set up the server, first edit package.json inside the attribute scripts and create the command npm run clusters, as shown here.

```
1   "scripts": {
2     "start": "npm run apidoc && babel-node index.js",
3     "clusters": "babel-node clusters.js",
4     "test": "NODE_ENV=test mocha test/**/*.js",
5     "apidoc": "apidoc -i routes/ -o public/apidoc"
6   }
```

Now, if you execute the command `npm run clusters`, this time the application will run distributed into the clusters. To make sure it worked, you will see the message `"NTask API - Port 3000"` more than once in the terminal, as shown in Figure 11-2.

Figure 11-2. Running Node.js in cluster mode

Basically, we have to load the module `cluster` and first, verify it is the master cluster via the `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), it does not fit in the conditional `if(cluster.isMaster)`. Therefore, 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 here.

- `listening`: Happens when a cluster is listening to a port. In this case, our application is listening to 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

Many things can be explored about developing clusters on Node.js. Here, we only applied a little bit, which was enough to run parallel processing. If you have to implement a more detailed cluster design, I recommend you read the documentation at https://nodejs. org/api/cluster.html.

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

```
1   "scripts": {
2     "start": "npm run apidoc && npm run clusters",
3     "clusters": "babel-node clusters.js",
4     "test": "NODE_ENV=test mocha test/**/*.js",
5     "apidoc": "apidoc -i routes/ -o public/apidoc"
6   }
```

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

Compacting Requests Using GZIP Middleware

To make the requests lighter and enable them to load faster, let's enable another middleware that will compact the JSON responses and the entire API documentation static files into GZIP format, a format compatible with several browsers. We'll do this simple, but important refactoring just using the module `compression`. First, let's install it.

```
1   npm install compression@1.6.1 --sav
```

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

```
1    import bodyParser from "body-parser";
2    import express from "express";
3    import morgan from "morgan";
4    import cors from "cors";
5    import compression from "compression";
6    import logger from "./logger.js";
7
8    module.exports = app => {
9      app.set("port", 3000);
10     app.set("json spaces", 4);
11     app.use(morgan("common", {
12       stream: {
13         write: (message) => {
14           logger.info(message);
15         }
16       }
17     }));
18     app.use(cors({
19       origin: ["http://localhost:3001"],
20       methods: ["GET", "POST", "PUT", "DELETE"],
21       allowedHeaders: ["Content-Type", "Authorization"]
22     }));
```

```
23    app.use(compression());
24    app.use(bodyParser.json());
25    app.use(app.auth.initialize());
26    app.use((req, res, next) => {
27      delete req.body.id;
28      next();
29    });
30    app.use(express.static("public"));
31  };
```

To test this implementation, restart the server and then go to the API documentation—after all, there are a lot of static files there—at the address http://localhost:3000/apidoc.

To see the details, open the browser console (Firefox and Google Chrome have great client-side consoles) and go to the Networks menu. There, you can see the transferred size versus file size, which will look something like Figure 11-3.
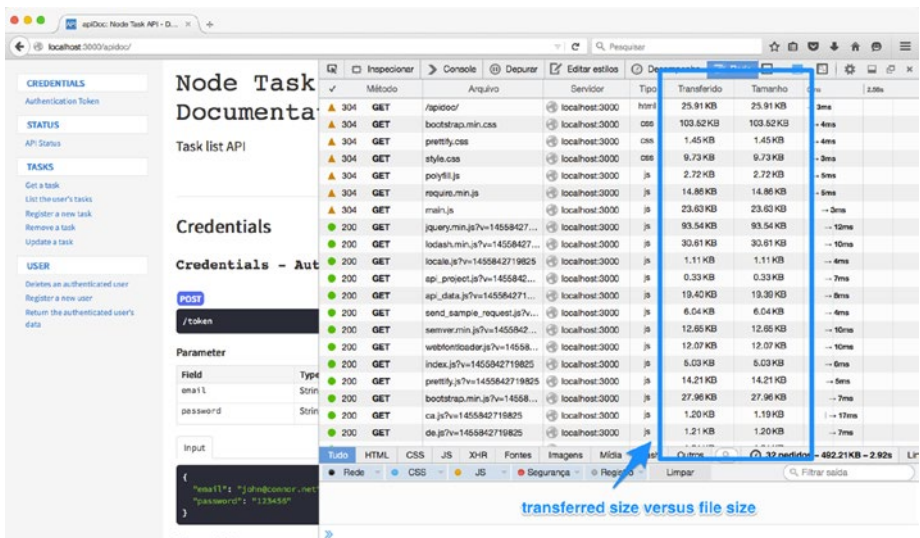
Figure 11-3. *Applying GZIP in some files*

Installing SSL Support to Use HTTPS

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

To implement an HTTPS protocol connection, it is necessary to buy a digital certificate for use in the production environment. For our purposes here, we are going to work with a fake certificate, which is not valid for production usage, but valid enough for didactic ends. To create a simple certificate, you can go to http://www.selfsignedcertificate.com.

Naviate to the `ntask` domain and click Generate. A new screen will be displayed with two extension files: `.key` and `.cert`. Download these files and put them into the root folder of your project.

Now, let's use the native `https` module to allow our server to start using the 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()` with the `https.createServer(credentials, app).listen()` function. To implement this change, edit the `libs/boot.js` as shown here.

```
1   import https from "https";
2   import fs from "fs";
3
4   module.exports = app => {
5     if (process.env.NODE_ENV !== "test") {
6     const credentials = {
7       key: fs.readFileSync("ntask.key", "utf8"),
8       cert: fs.readFileSync("ntask.cert", "utf8")
9     }
10    app.db.sequelize.sync().done(() => {
11      https.createServer(credentials, app)
12        .listen(app.get("port"), () => {
13          console.log(`NTask API - Port ${app.get("port")}`);
14        });
15      });
16    }
17  };
```

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 environment.

To test these changes, just restart and go to https://localhost:3000.

Armoring the API with Helmet

To complete 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 for treating the following HTTP settings:

- Configures the Content Security policy.

- Removes 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+.

- Disables client-side caching.

- Prevents sniffing attacks on the client Mime Type.

- Prevents ClickJacking attacks.

- Protects against Cross-Site Scripting (XSS) attacks.

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

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

To ensure maximum security on our API, we are going to use all middlewares provided by the helmet module, which can easily be included via the app. use(helmet())function. Edit the libs/middle- wares.js file as shown here.

```
1   import bodyParser from "body-parser";
2   import express from "express";
3   import morgan from "morgan";
4   import cors from "cors";
5   import compression from "compression";
6   import helmet from "helmet";
7   import logger from "./logger.js";
8
9   module.exports = app => {
10    app.set("port", 3000);
11    app.set("json spaces", 4);
12    app.use(morgan("common", {
13      stream: {
14        write: (message) => {
15          logger.info(message);
16        }
17      }
18    }));
19    app.use(helmet());
20    app.use(cors({
21      origin: ["http://localhost:3001"],
22      methods: ["GET", "POST", "PUT", "DELETE"],
23      allowedHeaders: ["Content-Type", "Authorization"]
24    }));
25    app.use(compression());
26    app.use(bodyParser.json());
27    app.use(app.auth.initialize());
28    app.use((req, res, next) => {
29      delete req.body.id;
30      next();
```

```
31     });
32     app.use(express.static("public"));
33   };
```

Now, restart your application and go to `https://localhost:3000`. Open the browser console and, on the Networks menu, you can view details of the GET / request's data. There, you'll see new items included in the header, as shown in Figure 11-4.
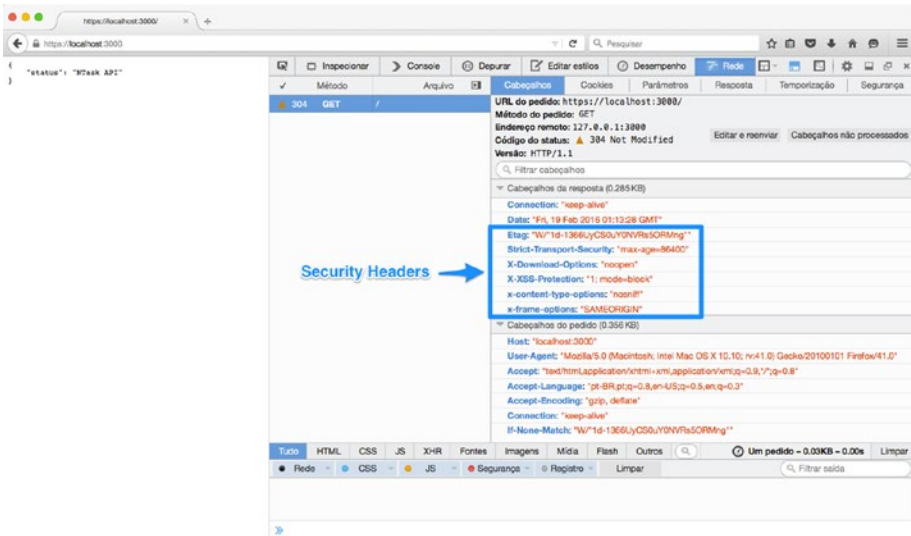
Figure 11-4. *Security headers*

Conclusion

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 that applies the main REST standards, has well-tested endpoints, persists data in a SQL database via the Sequelize.js module and, most important, follows best practices about performance and security for production environments.

The book is not complete yet, though. In the next chapter, we'll create a web application that is going to consume our API. It will be a simple single-page application (SPA) and will be developed using only JavaScript ES6 code.

CHAPTER 12

■ ■ ■

Building the Client-Side App: Part 1

After building a REST API using Node.js using JavaScript ES6 language, for the final two chapters of this book, we'll create a new project. This time it will be a front-end project using the best of JavaScript.

This book has devoted 80 percent of its content to back-end development, but now, we focus on the remaining 20percent, 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 two chapters, we are going to build an 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 perhaps I should say vanilla ES6.

Setting Up the App Environment

Our client-side app will be built under an object-oriented programming (OOP) 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 the `npm alias` command, like we used in the API project.
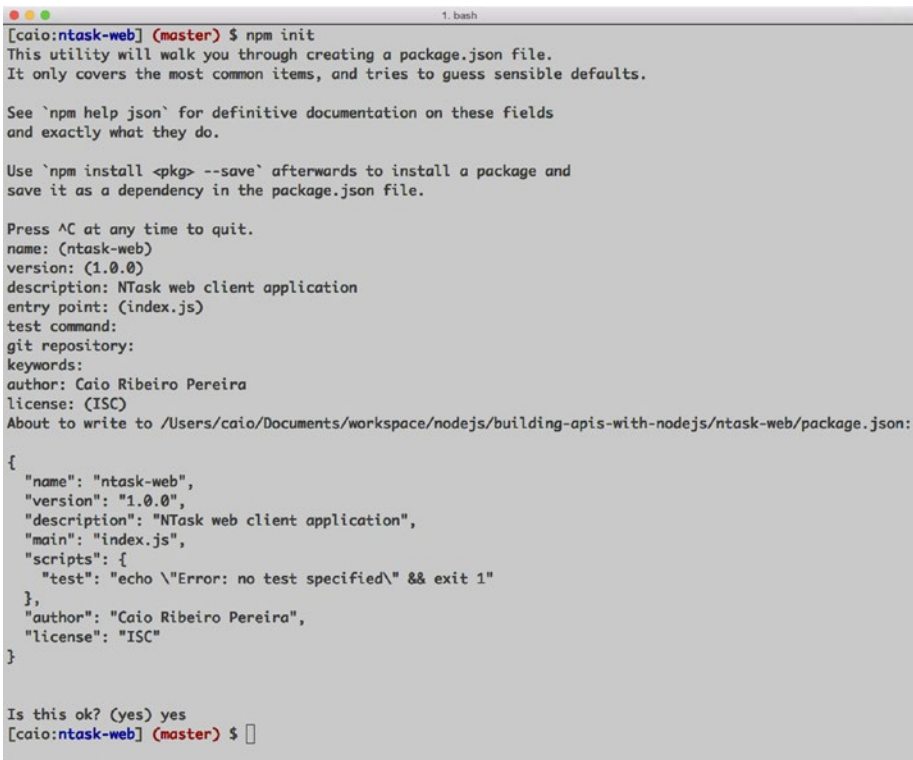
To kick off, let's open the terminal to create a new project, which cannot be in the same API directory. This new project will be named `ntask-web`.

```
1   mkdir ntask-web
2   cd ntask-web
3   npm init
```

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

After executing the `npm init` command, the file `package.json` is generated (see Figure 12-1). To organize this new project, we must create some directories in the project's root, as follows.

- **public**: For static files like CSS, fonts, JavaScript and images.

- **public/css**: For CSS files only (we use Ionic CSS).

- **public/fonts**: For font files (we use Ionic FontIcons).

- **public/js**: For JavaScript files, here we'll put a compiled and minified JavaScript file, generated from the **src** folder.

- **src/components**: For components business logic.

- **src/templates**: For components views.

```
1. bash
[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) $ []
```

Figure 12-1. *NTask Web generating* `package.json` *file*

To create these folders, just run this command.

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

Here is a list of all the modules that will be used in this project.

- • `http-server`: CLI for starting a simple HTTP server for static files only.

- • `browserify`: A JavaScript compiler that allows use of some NPM modules. We'll use some modules that have JavaScript Universal codes (codes that works in back-end and front-end layers) and also allows loading using CommonJS pattern, which is the same standard as Node.js.

- • `babelify`: A plug-in for the `browserify` module to enable reading and compiling 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`: A version of the `request` module focused for browsers, it is a cross-browser (compatible with all major browsers) and abstracts the whole complexity of an AJAX request.

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

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

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

Basically, we'll perform these tasks:

- • Compile and concatenate all ES6 codes from the `src` folder via the `npm run build` command.

- • Initiate the server in port `3001` via the `npm run server` command.

- • Create the command `npm start`, which will run the `npm run build` and `npm run server` commands.

To apply these alterations, edit the `package.json` file as shown here.

```
1   {
2     "name": "ntask-web",
3     "version": "1.0.0",
4     "description": "NTask web client application",
5       "scripts": {
6       "start": "npm run build && npm run server",
7       "server": "http-server public -p 3001",
8       "build": "browserify src -t babelify -o public/js/app.js"
9     },
```

```
10    "author": "Caio Ribeiro Pereira",
11    "dependencies": {
12      "babel-preset-es2015": "^6.5.0",
13      "babelify": "^7.2.0",
14      "browser-request": "^0.3.3",
15      "browserify": "^13.0.0",
16      "http-server": "^0.9.0",
17      "tiny-emitter": "^1.0.2"
18    }
19  }
```

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

```
1  {
2    "presets": ["es2015"]
3  }
```

After this setup, let's include some static files that will be responsible for the layout of our project. To save time, we 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 on the Angular framework. We only use the CSS file and font icon package. To do this, I recommend you download the files from the following links.

- Ionic CSS: http://code.ionicframework.com/1.0.0/css/ionic.min.css

- Ionic CSS icons: http://code.ionicframework.com/ionicons/2.0.0/css/ionicons.min.css

- Ionic Font icons: http://code.ionicframework.com/1.0.0/fonts/ionicons.eot

- Ionic Font icons: http://code.ionicframework.com/1.0.0/fonts/ionicons.svg

- Ionic Font icons: http://code.ionicframework.com/1.0.0/fonts/ionicons.ttf

- Ionic Font icons: http://code.ionicframework.com/1.0.0/fonts/ionicons.woff

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

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

```
 1   <!DOCTYPE html>
 2   <html>
 3     <head>
 4       <meta charset="utf-8">
 5       <title>NTask - Task manager</title>
 6       <meta name="viewport" content="width=device-width,initial-scale=1">
 7       <link rel="stylesheet" href="css/ionic.min.css">
 8       <link rel="stylesheet" href="css/ionicons.min.css">
 9       <script src="js/app.js" async defer></script>
10     </head>
11     <body>
12       <header class="bar bar-header bar-calm">
13         <h1 class="title">NTask</h1>
14       </header>
15       <div class="scroll-content ionic-scroll">
16         <div class="content overflow-scroll has-header">
17           <main></main>
18           <footer></footer>
19         </div>
20       </div>
21     </body>
22   </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. 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.

```
1   window.onload = () => {
2     alert("Welcome!");
3   };
```

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 http://localhost:3001. If everything works fine, you'll see the result shown in Figure 12-2.
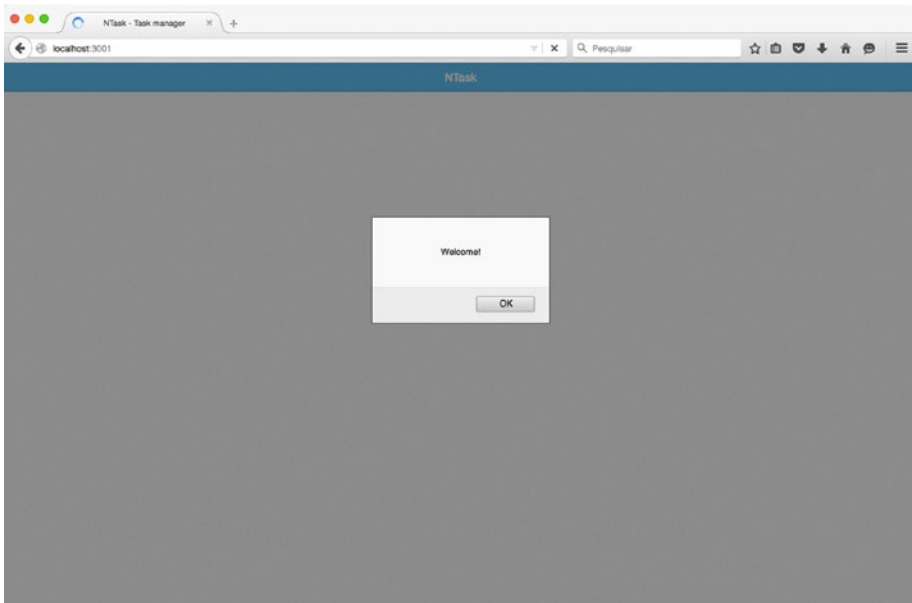
Figure 12-2. *First NTask screen*

Creating Sign-in and Signup Views

In this section, we 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 and only transfer data frequently between client and server.

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

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 users. All templates will be based on these API features.

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

```
1    console.log(`Olá ${nome}!`);
```

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

To understand this implementation better, let's write the sign-in home page templates using the function render() to return an HTML string. Create the file src/templates/signin.js as shown here.

```
1   exports.render = () => {
2     return `<form>
3       <div class="list">
4         <label class="item item-input item-stacked-label">
5           <span class="input-label">Email</span>
6           <input type="text" data-email>
7         </label>
8         <label class="item item-input item-stacked-label">
9           <span class="input-label">Password</span>
10            <input type="password" data-password>
11        </label>
12      </div>
13      <div class="padding">
14        <button class="button button-positive button-block">
15          <i class="ion-home"></i> Login
16        </button>
17      </div>
18    </form>
19    <div class="padding">
20      <button class="button button-block" data-signup>
21        <i class="ion-person-add"></i> Sign up
22      </button>
23    </div>`;
24  };
```

Now, to complete the flow, we also create the signup screen template. Create the file src/templates/signup.js, using this code.

```
1   exports.render = () => {
2     return `<form>
3       <div class="list">
4         <label class="item item-input item-stacked-label">
5           <span class="input-label">Name</span>
6           <input type="text" data-name>
7         </label>
8         <label class="item item-input item-stacked-label">
9           <span class="input-label">Email</span>
10          <input type="text" data-email>
11        </label>
12        <label class="item item-input item-stacked-label">
13          <span class="input-label">Password</span>
14          <input type="password" data-password>
15        </label>
16      </div>
```

```
17        <div class="padding">
18          <button class="button button-positive button-block">
19            <i class="ion-thumbsup"></i> Register
20          </button>
21        </div>
22      </form>`;
23    };
```

We now already have two important screens of the application. Now, we have to create the interaction code for these screens, which will be responsible for rendering these templates and, especially, interacting with the users and communicating with the API.

Writing Sign-in and Signup Components

The codes for the template's interaction are going to be put into the `src/components` folder, but before creating them, let's explore two new JavaScript ES6 features: *classes* and *inheritance.* To write our front-end code to be well organized, we'll create a parent class, which is going to have only two important attributes as inheritance for all components that 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 the `class NTask extends TinyEmitter` declaration), and these functionalities will be passed to their child classes as well, allowing our components to be able to listen and trigger events among the other components. To understand this class better, create the file `src/ntask.js` following this code.

```
1    import TinyEmitter from "tiny-emitter";
2    import request from "browser-request";
3
4    class NTask extends TinyEmitter {
5      constructor() {
6        super();
7        this.request = request;
8        this.URL = "https://localhost:3000";
9      }
10    }
11
12    module.exports = NTask;
```

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

Let's create our first component for the sign-in screen. All component 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 the

child class's constructor. We'll use the methods: `render()` (to render a template) and `addEventListener()` (to listen and treat any DOM components from the templates) to organize our components.

In this case, we will have two encapsulated events in the methods: `formSubmit()` (responsible for performing user authentication on the API) and `signupClick()` (responsible for rendering 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 a task that is going to be widely used is the template's transitions, which occurs when a user clicks on a template's button. The click event triggers the observer class to listen and delegate a task for a new template to be rendered in the browser.

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

```
1   import NTask from "../ntask.js";
2   import Template from "../templates/signin.js";
3
4   class Signin extends NTask {
5     constructor(body) {
6       super();
7       this.body = body;
8     }
9     render() {
10      this.body.innerHTML = Template.render();
11      this.body.querySelector("[data-email]").focus();
12      this.addEventListener();
13    }
14    addEventListener() {
15      this.formSubmit();
16      this.signupClick();
17    }
18    formSubmit() {
19      const form = this.body.querySelector("form");
20      form.addEventListener("submit", (e) => {
21        e.preventDefault();
22        const email = e.target.querySelector("[data-email]");
23        const password = e.target.querySelector("[data-password]");
24        const opts = {
25          method: "POST",
26          url: `${this.URL}/token`,
27          json: true,
28          body: {
29            email: email.value,
30            password: password.value
31          }
32        };
33        this.request(opts, (err, resp, data) => {
```

```
34          if (err || resp.status === 401) {
35             this.emit("error", err);
36          } else {
37             this.emit("signin", data.token);
38          }
39        });
40      });
41    }
42    signupClick() {
43      const signup = this.body.querySelector("[data-signup]");
44      signup.addEventListener("click", (e) => {
45        e.preventDefault();
46        this.emit("signup");
47      });
48    }
49  }
50
51  module.exports = Signin;
```

Let's also create the Signup class. This will follow the same pattern as the Signin class. To see how it should be, create the src/components/signup.js file as shown here.

```
1   import NTask from "../ntask.js";
2   import Template from "../templates/signup.js";
3
4   class Signup extends NTask {
5     constructor(body) {
6       super();
7       this.body = body;
8     }
9     render() {
10      this.body.innerHTML = Template.render();
11      this.body.querySelector("[data-name]").focus();
12      this.addEventListener();
13    }
14    addEventListener() {
15      this.formSubmit();
16    }
17    formSubmit() {
18      const form = this.body.querySelector("form");
19      form.addEventListener("submit", (e) => {
20        e.preventDefault();
21        const name = e.target.querySelector("[data-name]");
22        const email = e.target.querySelector("[data-email]");
23        const password = e.target.querySelector("[data-password]");
24        const opts = {
25          method: "POST",
26          url: `${this.URL}/users`,
```

```
27          json: true,
28          body: {
29            name: name.value,
30            email: email.value,
31            password: password.value
32          }
33        };
34        this.request(opts, (err, resp, data) => {
35          if (err || resp.status === 412) {
36            this.emit("error", err);
37          } else {
38            this.emit("signup", data);
39          }
40        });
41      });
42    }
43  }
44
45  module.exports = Signup;
```

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

```
1   import Signin from "./components/signin.js";
2   import Signup from "./components/signup.js";
3
4   class App {
5     constructor(body) {
6       this.signin = new Signin(body);
7       this.signup = new Signup(body);
8     }
9     init() {
10      this.signin.render();
11      this.addEventListener();
12    }
13    addEventListener() {
14      this.signinEvents();
15      this.signupEvents();
16    }
17    signinEvents() {
18      this.signin.on("error", () => alert("Authentication error"));
19      this.signin.on("signin", (token) => {
20        localStorage.setItem("token", `JWT ${token}`);
```

```
21          alert("You are logged in!");
22        });
23        this.signin.on("signup", () => this.signup.render());
24      }
25      signupEvents() {
26        this.signup.on("error", () => alert("Register error"));
27        this.signup.on("signup", (user) => {
28          alert(`${user.name} you were registered!`);
29          this.signin.render();
30        });
31      }
32    }
33
34    module.exports = App;
```

Basically, it created the component's events for successful authentication, authentication error, accessing the signup screen, successful signup, and signup error. The init() method starts the home page, which is the signin screen, and then executes the addEventListener() method to listen to all the component 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 the interactive flow of the application.

```
1    import App from "./app.js"
2
3    window.onload = () => {
4      const main = document.querySelector("main");
5      new App(main).init();
6    };
```

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

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

- *NTask API*: https://localhost:3000

- *Ntask Web*: http://localhost:3001

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

Next, add an exception in your browser to enable the API access. See Figure 12-3 to understand how to add an exception in the Mozilla Firefox browser, for example.
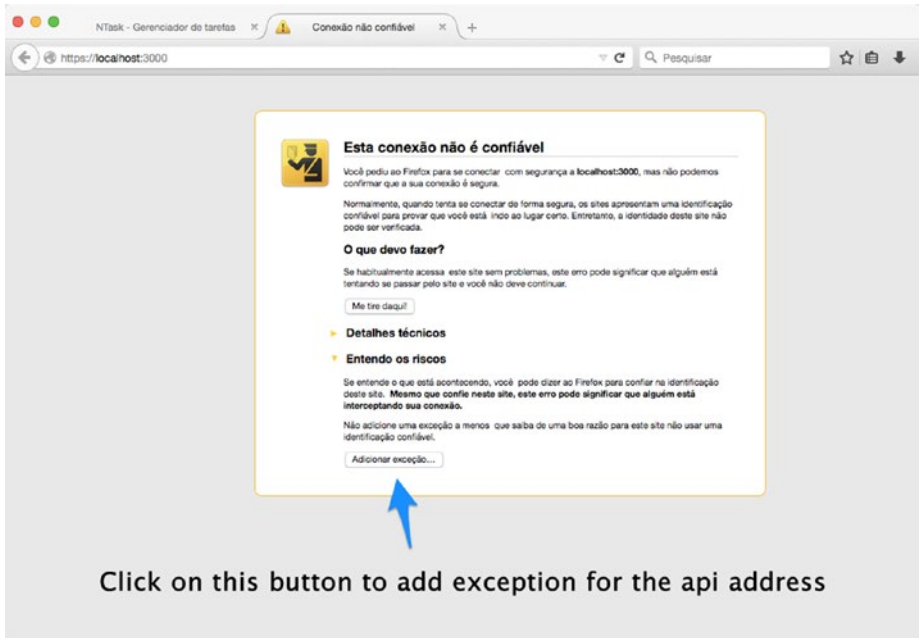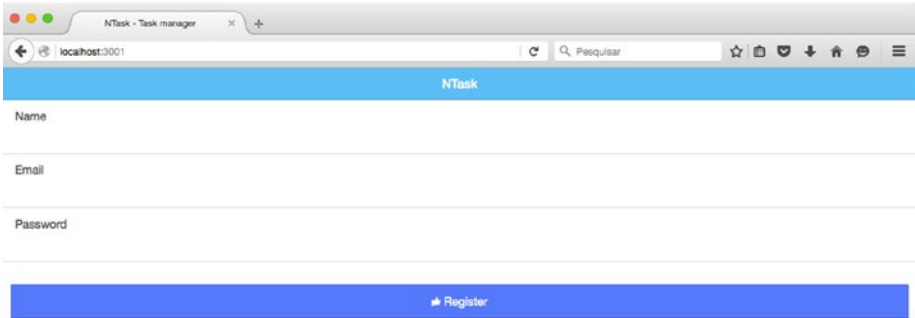
Figure 12-3. *Adding an exception for the API address*

Now that the API has full access, go to the NTask web address at http://localhost:3001.
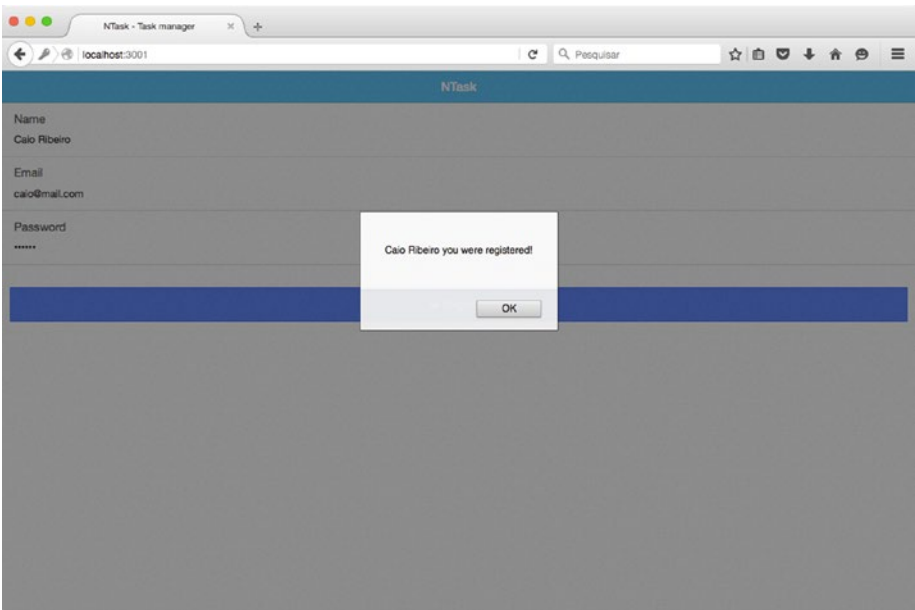
If everything is right, you'll be able to access the screens shown in Figures 12-4 through 12-7.
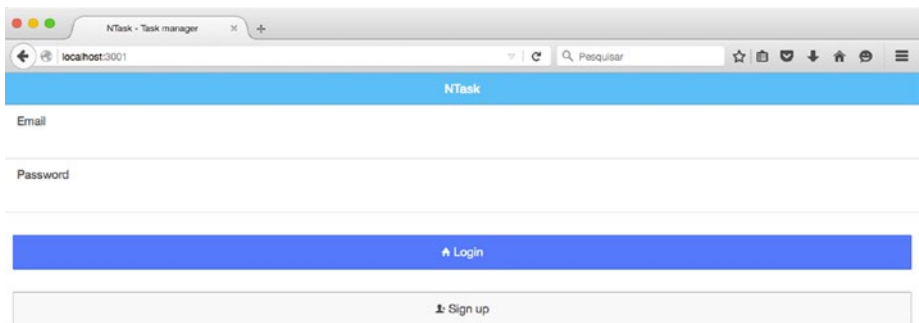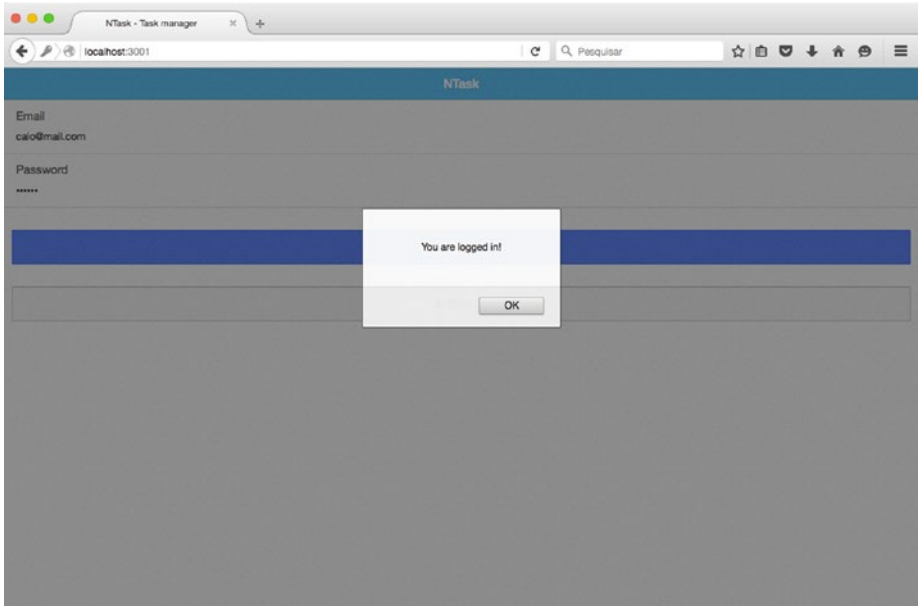

Figure 12-4. *Sign-in screen*

Figure 12-5. *User signup screen*

Figure 12-6. *Registering a new user*

Figure 12-7. *Logging in with a new account*

Conclusion

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

Building the Client-Side App: Part 2

Continuing with the client application construction, to this point we have an application with the main layout that connects with the API and allows us to authenticate a user to access the system. In this chapter, we are going to build the main features for users to 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, it will be great. This template must list all of the user's tasks, so this function will receive as an argument a tasks list. Using the `tasks.map()` function, it will generate a template's array of the tasks.

At 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 is encapsulated via the `renderTasks(tasks)` function. Otherwise, it displays a message about an empty task list.

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

```
1   const renderTasks = tasks => {
2     return tasks.map(task => {
3       let done = task.done ? "ios-checkmark" : "ios-circle-outline";
4       return `<li class="item item-icon-left item-button-right">
5         <i class="icon ion-${done}" data-done
6           data-task-done="${task.done ? 'done' : ''}"
7           data-task-id="${task.id}"></i>
8         ${task.title}
9         <button data-remove data-task-id="${task.id}"
10          class="button button-assertive">
11          <i class="ion-trash-a"></i>
12        </button>
13      </li>`;
14    }).join("");
```

```
15    };
16    exports.render = tasks => {
17      if (tasks && tasks.length) {
18        return `<ul class="list">${renderTasks(tasks)}</ul>`;
19      }
20      return `<h4 class="text-center">The task list is empty</h4>`;
21    };
```

Using the attributes data-task-done, data-task-id, data-done, and data-remove, we 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())or setting which task will be performed (via method taskDoneCheckbox()). These business rules will be written in the src/components/tasks.js file.

```
1     import NTask from "../ntask.js";
2     import Template from "../templates/tasks.js";
3
4     class Tasks extends NTask {
5       constructor(body) {
6         super();
7         this.body = body;
8       }
9       render() {
10        this.renderTaskList();
11      }
12      addEventListener() {
13        this.taskDoneCheckbox();
14        this.taskRemoveClick();
15      }
16      renderTaskList() {
17        const opts = {
18          method: "GET",
19          url: `${this.URL}/tasks`,
20          json: true,
21          headers: {
22            authorization: localStorage.getItem("token")
23          }
24        };
25        this.request(opts, (err, resp, data) => {
26          if (err) {
27            this.emit("error", err);
28          } else {
29            this.body.innerHTML = Template.render(data);
30            this.addEventListener();
31          }
32        });
33      }
34      taskDoneCheckbox() {
```

```
35      const dones = this.body.querySelectorAll("[data-done]");
36      for(let i = 0, max = dones.length; i < max; i++) {
37          dones[i].addEventListener("click", (e) => {
38              e.preventDefault();
39              const id = e.target.getAttribute("data-task-id");
40              const done = e.target.getAttribute("data-task-done");
41              const opts = {
42                method: "PUT",
43                url: `${this.URL}/tasks/${id}`,
44                headers: {
45                  authorization: localStorage.getItem("token"),
46                  "Content-Type": "application/json"
47                },
48                body: JSON.stringify({done: !done})
49              };
50              this.request(opts, (err, resp, data) => {
51                if (err || resp.status === 412) {
52                  this.emit("update-error", err);
53                } else {
54                  this.emit("update");
55                }
56              });
57          });
58      }
59      }
60      taskRemoveClick() {
61        const removes = this.body.querySelectorAll("[data-remove]");
62        for(let i = 0, max = removes.length; i < max; i++) {
63          removes[i].addEventListener("click", (e) => {
64              e.preventDefault();
65              if (confirm("Do you really wanna delete this task?")) {
66                const id = e.target.getAttribute("data-task-id");
67                const opts = {
68                  method: "DELETE",
69                  url: `${this.URL}/tasks/${id}`,
70                  headers: {
71                    authorization: localStorage.getItem("token")
72                  }
73                };
74                this.request(opts, (err, resp, data) => {
75                  if (err || resp.status === 412) {
76                    this.emit("remove-error", err);
77                  } else {
78                    this.emit("remove");
79                  }
80                });
81              }
82          });
```

```
83        }
84      }
85    }
86
87    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.

```
1   exports.render = () => {
2     return `<form>
3       <div class="list">
4         <label class="item item-input item-stacked-label">
5           <span class="input-label">Task</span>
6           <input type="text" data-task>
7         </label>
8       </div>
9       <div class="padding">
10        <button class="button button-positive button-block">
11          <i class="ion-compose"></i> Add
12        </button>
13      </div>
14    </form>`;
15  };
```

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.

```
1   import NTask from "../ntask.js";
2   import Template from "../templates/taskForm.js";
3
4   class TaskForm extends NTask {
5     constructor(body) {
6       super();
7       this.body = body;
8     }
9     render() {
10      this.body.innerHTML = Template.render();
11      this.body.querySelector("[data-task]").focus();
12      this.addEventListener();
13    }
14    addEventListener() {
15      this.formSubmit();
16    }
```

```
17    formSubmit() {
18      const form = this.body.querySelector("form");
19      form.addEventListener("submit", (e) => {
20        e.preventDefault();
21        const task = e.target.querySelector("[data-task]");
22        const opts = {
23          method: "POST",
24          url: `${this.URL}/tasks`,
25          json: true,
26          headers: {
27            authorization: localStorage.getItem("token")
28          },
29          body: {
30            title: task.value
31          }
32        };
33        this.request(opts, (err, resp, data) => {
34          if (err || resp.status === 412) {
35            this.emit("error");
36          } else {
37            this.emit("submit");
38          }
39        });
40      });
41    }
42  }
43
44  module.exports = TaskForm;
```

Views and Components for Logged Users

To finish the screen creation of our application, we build the last screen, which will display the logged user's data and a button to be allow the user to cancel his or her account. This screen will have a component that is 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 file.

```
1   exports.render = user => {
2     return `<div class="list">
3       <label class="item item-input item-stacked-label">
4         <span class="input-label">Name</span>
5         <small class="dark">${user.name}</small>
6       </label>
7       <label class="item item-input item-stacked-label">
8         <span class="input-label">Email</span>
9         <small class="dark">${user.email}</small>
10      </label>
```

```
11      </div>
12      <div class="padding">
13        <button data-remove-account
14          class="button button-assertive button-block">
15          <i class="ion-trash-a"></i> Cancel account
16        </button>
17      </div>`;
18    };
```

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

```
1    import NTask from "../ntask.js";
2    import Template from "../templates/user.js";
3
4    class User extends NTask {
5      constructor(body) {
6        super();
7        this.body = body;
8      }
9      render() {
10        this.renderUserData();
11      }
12      addEventListener() {
13        this.userCancelClick();
14      }
15      renderUserData() {
16        const opts = {
17          method: "GET",
18          url: `${this.URL}/user`,
19          json: true,
20          headers: {
21            authorization: localStorage.getItem("token")
22          }
23        };
24        this.request(opts, (err, resp, data) => {
25          if (err || resp.status === 412) {
26            this.emit("error", err);
27          } else {
28            this.body.innerHTML = Template.render(data);
29            this.addEventListener();
30          }
31        });
32      }
33      userCancelClick() {
34        const button = this.body.querySelector("[data-remove-account]");
35        button.addEventListener("click", (e) => {
36          e.preventDefault();
```

```
37          if (confirm("This will cancel your account, are you sure?")) {
38            const opts = {
39              method: "DELETE",
40              url: `${this.URL}/user`,
41              headers: {
42                authorization: localStorage.getItem("token")
43              }
44            };
45            this.request(opts, (err, resp, data) => {
46              if (err || resp.status === 412) {
47                this.emit("remove-error", err);
48              } else {
49                this.emit("remove-account");
50              }
51            });
52          }
53        });
54      }
55    }
56
57    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 user 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.

```
1    exports.render = path => {
2      let isTasks = path === "tasks" ? "active" : "";
3      let isTaskForm = path === "taskForm" ? "active" : "";
4      let isUser = path === "user" ? "active" : "";
5      return `
6        <div class="tabs-striped tabs-color-calm">
7          <div class="tabs">
8            <a data-path="tasks" class="tab-item ${isTasks}">
9              <i class="icon ion-home"></i>
10           </a>
11           <a data-path="taskForm" class="tab-item ${isTaskForm}">
12             <i class="icon ion-compose"></i>
13           </a>
14           <a data-path="user" class="tab-item ${isUser}">
15             <i class="icon ion-person"></i>
16           </a>
17           <a data-logout class="tab-item">
18             <i class="icon ion-android-exit"></i>
```

```
19              </a>
20            </div>
21         </div>`;
22    };
```

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

```
1    import NTask from "../ntask.js";
2    import Template from "../templates/footer.js";
3
4    class Menu extends NTask {
5      constructor(body) {
6        super();
7        this.body = body;
8      }
9      render(path) {
10       this.body.innerHTML = Template.render(path);
11       this.addEventListener();
12     }
13     clear() {
14       this.body.innerHTML = "";
15     }
16     addEventListener() {
17       this.pathsClick();
18       this.logoutClick();
19     }
20     pathsClick() {
21       const links = this.body.querySelectorAll("[data-path]");
22       for(let i = 0, max = links.length; i < max; i++) {
23         links[i].addEventListener("click", (e) => {
24           e.preventDefault();
25           const link = e.target.parentElement;
26           const path = link.getAttribute("data-path");
27           this.emit("click", path);
28         });
29       }
30     }
31     logoutClick() {
32       const link = this.body.querySelector("[data-logout]");
33       link.addEventListener("click", (e) => {
34         e.preventDefault();
35         this.emit("logout");
36       })
37     }
38   }
39
40   module.exports = Menu;
```

Treating All Screen 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` file, applying this simple modification.

```
1   import App from "./app.js";
2
3   window.onload = () => {
4     const main = document.querySelector("main");
5     const footer = document.querySelector("footer");
6     new App(main, footer).init();
7   };
```

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 treating the events of each component. By making this change we will ensure the correct flow for all screen transitions, the menu transition, and the data traffic between the `ntask-api` and `ntask-web` projects. To do this, edit the `src/app.js` script.

```
1    import Tasks from "./components/tasks.js";
2    import TaskForm from "./components/taskForm.js";
3    import User from "./components/user.js";
4    import Signin from "./components/signin.js";
5    import Signup from "./components/signup.js";
6    import Menu from "./components/menu.js";
7
8    class App {
9      constructor(body, footer) {
10       this.signin = new Signin(body);
11       this.signup = new Signup(body);
12       this.tasks = new Tasks(body);
13       this.taskForm = new TaskForm(body);
14       this.user = new User(body);
15       this.menu = new Menu(footer);
16     }
17     init() {
18       this.signin.render();
19       this.addEventListener();
20     }
21     addEventListener() {
22       this.signinEvents();
23       this.signupEvents();
24       this.tasksEvents();
25       this.taskFormEvents();
```

```
26        this.userEvents();
27        this.menuEvents();
28      }
29      signinEvents() {
30        this.signin.on("error", () => alert("Authentication error"));
31        this.signin.on("signin", (token) => {
32          localStorage.setItem("token", `JWT ${token}`);
33          this.menu.render("tasks");
34          this.tasks.render();
35        });
36        this.signin.on("signup", () => this.signup.render());
37      }
38      signupEvents(){
39        this.signup.on("error", () => alert("Register error"));
40        this.signup.on("signup", (user) => {
41          alert(`${user.name} you were registered!`);
42          this.signin.render();
43        });
44      }
45      tasksEvents() {
46        this.tasks.on("error", () => alert("Task list error"));
47        this.tasks.on("remove-error", () => alert("Task delete error"));
48        this.tasks.on("update-error", () => alert("Task update error"));
49        this.tasks.on("remove", () => this.tasks.render());
50        this.tasks.on("update", () => this.tasks.render());
51      }
52      taskFormEvents() {
53        this.taskForm.on("error", () => alert("Task register error"));
54        this.taskForm.on("submit", () => {
55          this.menu.render("tasks");
56          this.tasks.render();
57        });
58      }
59      userEvents() {
60        this.user.on("error", () => alert("User load error"));
61        this.user.on("remove-error", () => alert("Cancel account error"));
62        this.user.on("remove-account", () => {
63          alert("So sad! You are leaving us :(");
64          localStorage.clear();
65          this.menu.clear();
66          this.signin.render();
67        });
68      }
69      menuEvents() {
70        this.menu.on("click", (path) => {
71          this.menu.render(path);
72          this[path].render();
73        });
```

```
74        this.menu.on("logout", () => {
75          localStorage.clear();
76          this.menu.clear();
77          this.signin.render();
78        })
79      }
80    }
81
82    module.exports = App;
```

Phew! It's done! We built our simple but useful client application to interact with our current API. To test it, you just need to restart the client application and use it normally. Figures 13-1 through 13-3 display some new screens to access.
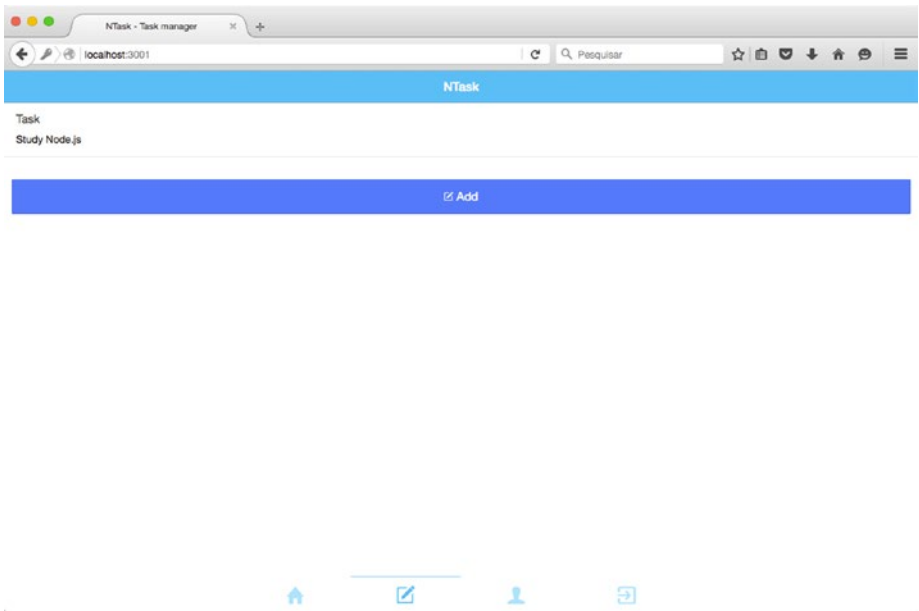
Figure 13-1. *Adding a task*
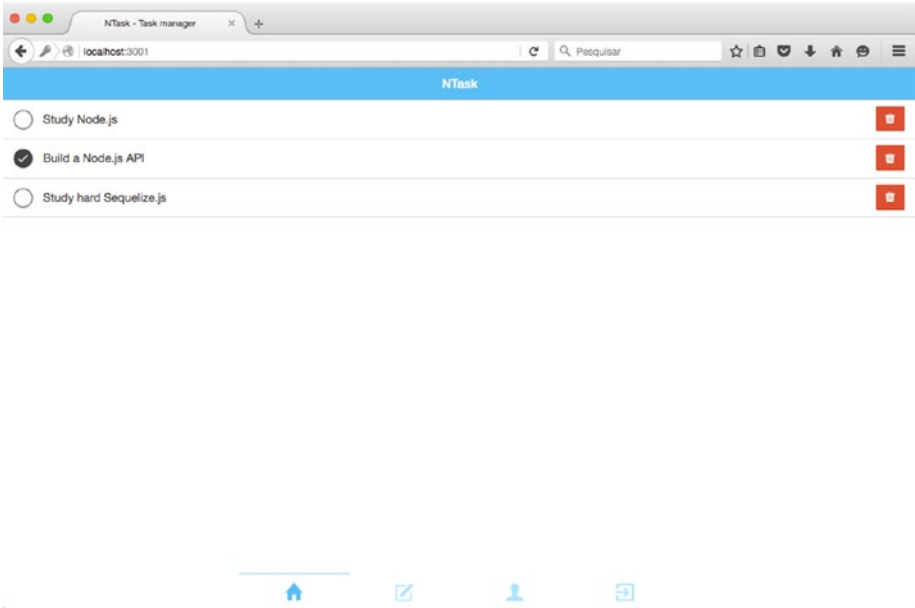
Figure 13-2. *Listing and checking some tasks*
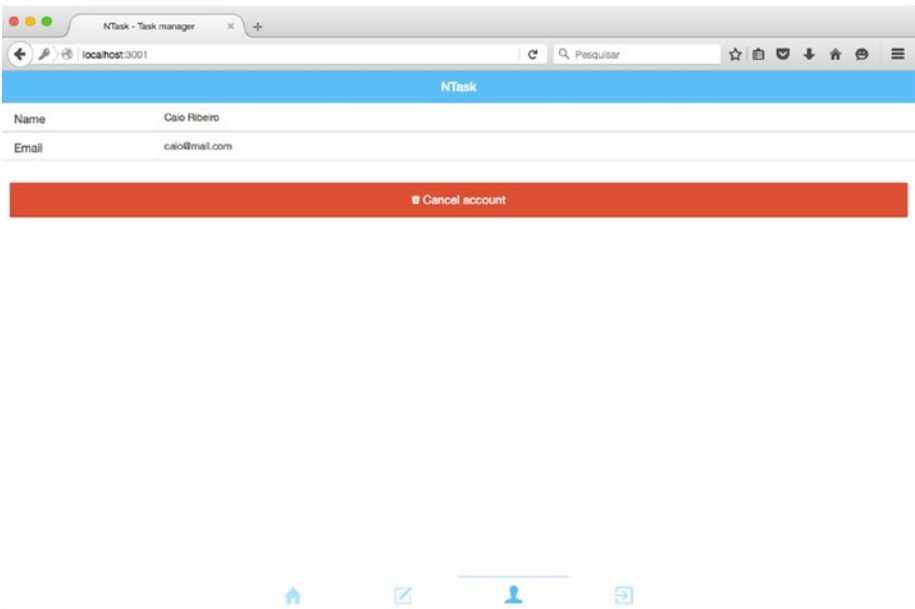
Figure 13-3. *User settings screen*

Conclusion

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

Remember that all source code is available on my personal GitHub. Just access this link: `github.com/caioribeiro-pereira/building-apis-with-nodejs`.

Thank you very much for reading this book!

Index