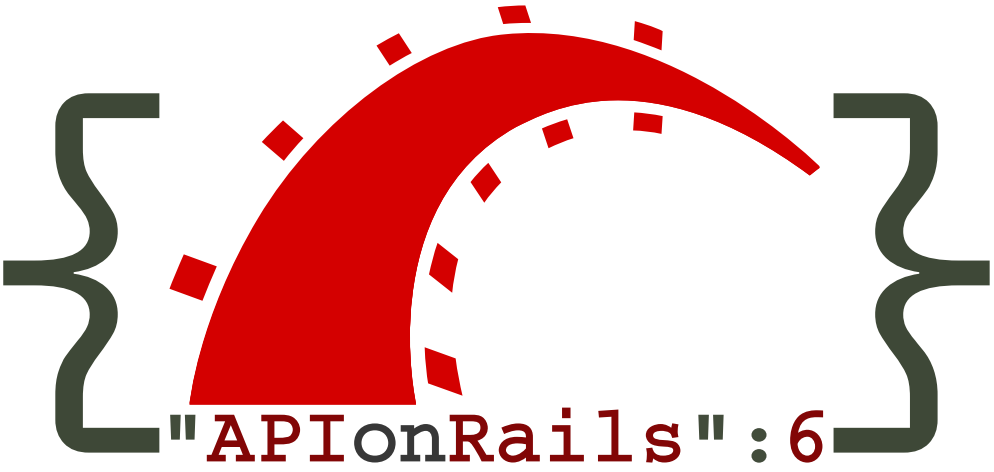
Alexandre Rousseau



API on Rails 6

Alexandre Rousseau

Version 6.9, 2020-12-20

Table of Contents

Before

Foreword

"API on Rails 6" is based on "APIs on Rails: Building REST APIs with Rails". It was initially published in 2014 by Abraham Kuri under the licenses MIT and Beerware.

The first version was not maintained and was initially planned for Ruby on Rails version 4, which does not receive more than security updates. I wanted to update this excellent book, adapting it to new versions of Ruby on Rails. Therefore, this book is available for Ruby on Rails versions 5.2 and 6.0 (the one you are currently reading).

NOTE	This book is also available in the Molière language (It means french).

About the author

My name is Alexandre Rousseau, and I am a Rails developer with more than 4 years of experience (at the time of writing). I am currently a partner in a company (iSignif) to build and maintain a SAAS product using Rails. I also contribute to the Ruby community by producing and maintaining some gems that you can consult on my Rubygems.org profile. Most of my projects are on GitHub, then don't hesitate to follow me.

This book's source code is available in Asciidoctor format on GitHub. Feel free to fork the project if you want to improve it or fix mistakes I didn't notice.

Copyright and license

This book is provided on MIT license. All the book's source code is available on Markdown format on GitHub

MIT license

This book's cover picture uses a beautiful photo shot by Yoann Siloine who published it on Unsplash.

Thanks

A big "thank you" to all Github contributors who keep this book alive. In alphabetical order:

- airdry
- Landris18
- lex111
- cuilei5205189
- franklinjosmell
- notapatch
- promisepreston
- tacataca

Introduction

Welcome to API on Rails 6, a tutorial on steroids to learn the best way to build your next API with Rails. The purpose of this book is to provide a comprehensive methodology to develop a RESTful API following best practices.

As soon as you finish this book, you will be able to create your own API and integrate it with any client, such as a web browser or mobile application. The generated code is built with Ruby on Rails 6.0, which is the current version.

The purpose of this book is not only to teach you how to build an API with Rails but rather to teach you how to build an **evolutive** and **maintainable** API with Rails. That is, improve your current knowledge with Rails. On this journey, you will learn to:

- Use Git for version control
- Building JSON responses
- Test your end-points with unit and functional tests
- Set up authentication with JSON Web Tokens (JWT)
- Use JSON:API specification
- Optimize and cache the API

I strongly recommend you follow all the steps in this book. Try not to skip chapters because I will give you some tips and tricks to improve your skills throughout the book. You can consider yourself the main character of a video game that gains a level in each chapter.

In the first chapter, I will explain how to configure your environment (if you don't already have it). Then we will create an application called `market_place_api`. I will ensure that I teach you the best practices I have learned during my experience. This means that we'll start using **Git** just after initializing the project.

We'll build the application following a simple working method that I use daily in the next chapters. We will develop the entire application using Test Driven Development (TDD). I will also explain the interest of using an API for your next project and choosing a suitable response format such as JSON or XML. Further on, we will get our hands on the code and complete the application's basics by building all the necessary roads. We will also secure access to the API by building authentication by exchanging HTTP headers. Finally, in the last chapter, we will add some optimization techniques to improve the

server’s structure and response times.

The final application will scratch the surface of being a market place where users will be able to place orders, upload products, and more. There are plenty of options out there to set up an online store, such as Shopify, Spree, or Magento.

Conventions on this book

The conventions in this book are based on the ones from Ruby on Rails Tutorial. In this section, I'll mention some that may not be so clear.

I'll be using many examples using command-line instructions. I won't deal with windows cmd (sorry guys), so all the examples use Unix-style command line prompt, as follows:

```
$ echo "A command-line command"
A command-line command
```

I'll be using some guidelines related to the language. What I mean by this is:

- **Avoid** means you are not supposed to do it
- **Prefer** indicates that from the 2 options, the first it's a better fit
- **Use** means you are good to use the resource

If for any reason you encounter some errors when running a command, rather than trying to explain every possible outcome, I recommend you to `google it', which I don't consider a bad practice or whatsoever. But if you feel like you want to grab a beer or have some trouble with the tutorial, you can always email me.

Development environments

One of the most painful parts for almost every developer is setting everything up, but as long as you get it done, the next steps should be a piece of cake and well rewarded. So I will guide you to keep you motivated.

Text editors and Terminal

There are many cases in which development environments may differ from computer to computer. That is not the case with text editors or IDE's. I think for Rails development an IDE is way too much, but some other might find that the best way to go, so if that it's your case I recommend you go with RadRails or RubyMine, both are well supported and come with many integrations out of the box.

- **Text editor**: I personally use vim as my default editor with janus, which will add and handle many of the plugins you are probably going to use. In case you are not a *vim* fan like me, there are a lot of other solutions such as Sublime Text which is a cross-platform easy to learn and customize (this is probably your best option), it is highly inspired by TextMate (only available for Mac OS). A third option uses a more recent text editor from the guys at GitHub called Atom. It's a promising text editor made with JavaScript. It is easy to extend and customize to meet your needs. Give it a try. Any of the editors I present will do the job, so I'll let you decide which one fits your eye.
- **Terminal**: If you decided to go with kaishi for setting the environment, you would notice that it sets the default shell to zsh, which I highly recommend. For the terminal, I'm not a fan of the *Terminal* app that comes out of the box if you are on Mac OS, so check out iTerm2, which is a terminal replacement for Mac OS. If you are on Linux, you probably have a nice terminal already, but the default should work fine.

Browsers

When it comes to browsers, I would say Firefox immediately, but some other developers may say Chrome or even Safari. Any of those will help you build the application you want. They come with a nice inspector not just for the DOM but also for network analysis and many other features you might know already.

Package manager

- **Mac OS**: There are many options to manage how you install packages on your Mac, such as Mac Ports or Homebrew, both are good options, but I would choose the last one, I've encountered fewer troubles when I install software, and I manage it. To install brew, just run the command below:

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/inst
all)"
```

- **Linux**: You are all set! It really does not matter if you are using apt, pacman, yum as long you feel comfortable with it, and you know how to install packages so you can keep moving forward.

Git

We will be using Git a lot, and you should use it too, not just for the purpose of this tutorial but for every single project.

- on Mac OS: `$ brew install git`
- on Linux: `$ sudo apt-get install git`

Ruby

There are many ways in which you can install and manage ruby, and by now, you should probably have some version installed if you are on Mac OS. To see which version you have, just type:

```
$ ruby -v
```

Rails 6.0 requires the installation of version 2.5 or higher.

I recommend using Ruby Version Manager (RVM) or rbenv to install it. We will use RVM in this tutorial, but it doesn't matter which of these two options you use.

The principle of these tools is allowing you to install several versions of Ruby on the same machine, in an environment that is airtight to a possible version installed on your operating system, and to be able to switch from one to the other easily.

To install RVM, go to https://rvm.io/ and install the GPG footnote key:[The GPG key allows you to verify the identity of the author of the sources you download.]. Once that's done:

```
$ gpg --keyserver hkp://keys.gnupg.net --recv-keys
409B6B1796C275462A1703113804BB82D39DC0E3
7D2BAF1CF37B13E2069D6956105BD0E739499BDB
$ \curl -sSL https://get.rvm.io | bash
```

Next, it is time to install ruby:

```
$ rvm install 2.6
```

Now it is time to install the rest of the dependencies we will be using.

Gems, Rails & Missing libraries

First, we update the gems on the whole system:

```
$ gem update --system
```

In some cases, if you are on a Mac OS, you will need to install some extra libraries:

```
$ brew install libtool libxslt libksba openssl
```

We then install the necessary gems and ignore documentation for each gem:

```
$ gem install bundler
$ gem install rails -v 6.0.0
```

Check for everything to be running nice and smooth:

```
$ rails -v
Rails 6.0.0
```

Database

I highly recommend you install Postgresql to manage your databases. But here, we'll be using SQlite for simplicity. If you are using Mac OS, you should be ready to go. In case you are on Linux, don't worry. We have you covered:

```
$ sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev
```

or

```
$ sudo yum install libxslt-devel libxml2-devel libsqlite3-devel
```

Initializing the project

Initializing a Rails application may be pretty straightforward for you. If that is not the case, here is a super quick tutorial.

There is the command:

```
$ mkdir ~/workspace
$ cd ~/workspace
$ rails new market_place_api --api
```

> **NOTE** The `--api` option appeared in version 5 of Rails. It allows you to limit the libraries and *Middleware* included in the application. This also avoids generating HTML views when using Rails generators.

As you may guess, the commands above will generate the bare bones of your Rails application.

Versioning

Remember that Git helps you track and maintain your code history. Keep in mind that the source code of the application is published on GitHub. You can follow the project on GitHub.

Ruby on Rails initialized the Git directory for you when you used the `rails new` command. This means that you do not need to execute the `git init` command.

However, it is necessary to configure the information of the author of *commits*. If you have not already done so, go to the directory and run the following commands:

```
$ git config --global user.name "Type in your name"
$ git config --global user.email "Type in your email"
```

Rails also provide a *.gitignore* file to ignore some files that we don't want to track. The default *.gitignore* file should look like the one shown below:

.gitignore

```
# Ignore bundler config.
/.bundle

# Ignore the default SQLite database.
/db/*.sqlite3
/db/*.sqlite3-journal

# Ignore all logfiles and tempfiles.
/log/*
/tmp/*
!/log/.keep
!/tmp/.keep

# Ignore uploaded files in development.
/storage/*
!/storage/.keep
.byebug_history

# Ignore master key for decrypting credentials and more.
/config/master.key
```

After modifying the *.gitignore* file, we just need to add the files and commit the changes, the necessary commands are shown below:

```
$ git add .
$ git commit -m "Initial commit"
```

> **TIP** I have found that committing a message starting with a present tense verb, describing what the commit does and not what it did, helps when you are exploring the history of the project. I find it is more natural to read and understand. I'll follow this practice until the end of the tutorial.

Lastly and as an optional step, we setup the GitHub (I'm not going through that here) project and push our code to the remote server: We first add the remote:

```
$ git remote add origin
git@github.com:madeindjs/market_place_api_6.git
```

Then we push the code:

```
$ git push -u origin master
```

As we move forward with the tutorial, I'll be using the practices I follow daily. This includes working with branches, rebasing, squash and some more. For now, you don't have to worry if some of these don't sound familiar to you. I walk you through them in time.

Conclusion

It’s been a long way through this chapter. If you reach here, let me congratulate you and be sure that things will get better from this point. So let’s get our hands dirty and start typing some code!

The API

In this section, I'll outline the application. By now, you should have read the previous chapter. If you did not read it, I recommend you to do it.

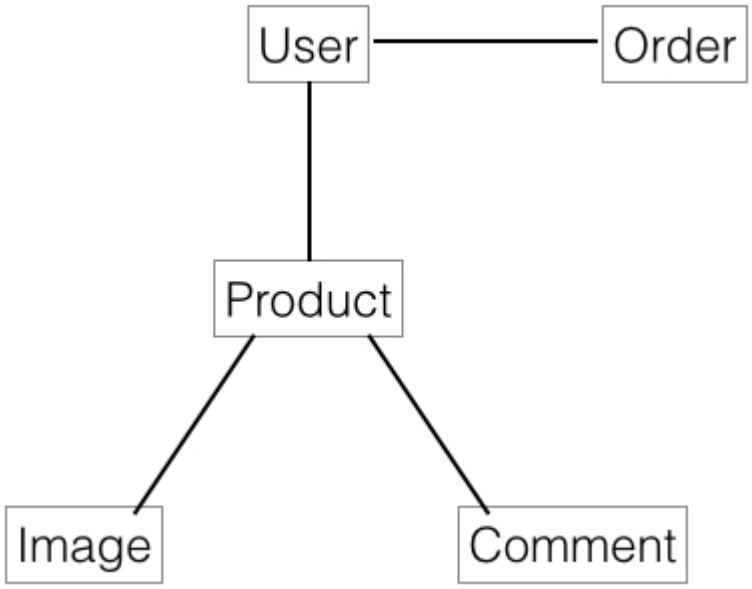
You can clone the project until this point with:

```
$ git checkout tags/checkpoint_chapter02
```

To summarize, we simply generated our Rails application and made our first commit.

Planning the application

As we want to go simple with the application, it consists of five models. Don't worry if you don't fully understand what is going on. We will review and build each of these resources as we move on with the tutorial.



In brief, the user will be able to place many orders, upload multiple products which can have many images or comments from other users on the app.

We will not build views for displaying or interacting with the API, so not to make this a huge tutorial, I'll let that to you. There are plenty of options out there like javascript frameworks (Angular, Vue.js, React.js).

By this point, you must be asking yourself:

> all right, but I need to explore or visualize the API we will be building?

That's fair. Probably if you google something related to API exploring, an application called Postman will pop. It is great software, but we won't be using that anyway because we'll use **cURL**, allowing anybody to reproduce requests on any computer.

Setting the API

An API is defined by wikipedia as *an application programming interface (API) specifies how some software components should interact with each other.* In other words, the way systems interact with each other through a common interface, in our case, a web service built with JSON. There are other kinds of communication protocols like SOAP, but we are not covering that here.

As the Internet media type standard, JSON is widely accepted, readable, extensible, and easy to implement. Many of the current frameworks consume JSON APIs by default (Angular or Vue.js, for example). There are also great libraries for Objective-C too like AFNetworking or RESTKit. There are probably good solutions for Android, but I might not be the right person to recommend something because of my lack of experience in that development platform.

All right. So we are building our API with JSON. There are many ways to achieve this. The first thing that comes to mind would be just to start adding routes defining the endpoints. This may be bad because they may not have a URI pattern clear enough to know which resource is being exposed. The protocol or structure I'm talking about is REST which stands for Representational State Transfer and by Wikipedia definition

```
aService.getUser("1")
```

And in REST you may call a URL with a specific HTTP request, in this case with a GET request: http://domain.com/resources_name/uri_pattern

RESTful APIs must follow at least three simple guidelines:

- A base URI, such as http://example.com/resources/.
- An Internet media type to represent the data is commonly JSON and is commonly set through header exchange.
- Follows the standard HTTP Methods such as GET, POST, PUT, DELETE.
  - **GET**: Reads the resource or resources defined by the URI pattern
  - **POST**: Creates a new entry into the resources collection
  - **PUT**: Updates a collection or member of the resources
  - **DELETE**: Destroys a collection or member of the resources

This might not be clear enough or may look like a lot of information to digest, but hopefully, it'll get a lot easier to understand as we

move on with the tutorial.

Routes, Constraints and Namespaces

Before start typing any code, we prepare the code with git. We'll be using a branch per chapter, upload it to GitHub and then merge it on master branch. So let's get started. Open the terminal, cd to the market_place_api directory and type in the following:

```
$ git checkout -b chapter02
Switched to a new branch 'chapter02'
```

We will only be working on the config/routes.rb, as we are just going to set the constraints and the default response format for each request.

config/routes.rb

```
Rails.application.routes.draw do
  # ...
end
```

First of all, erase all commented code that comes within the file. We are not gonna need it. Then commit it, just as a warm-up:

```
$ git add config/routes.rb
$ git commit -m "Removes comments from the routes file"
```

We are going to isolate the API controllers under a namespace. With Rails, this is fairly simple: you just have to create a folder under the app/controllers named API. The name is important because that's the namespace we'll use for managing the controllers for the API endpoints.

```
$ mkdir app/controllers/api
```

Then we add that namespace into our *routes.rb* file:

config/routes.rb

```
Rails.application.routes.draw do
  # API definition
  namespace :api do
    # We are going to list our resources here
  end
end
```

By defining a namespace under the routes.rb file. Rails will automatically map that namespace to a directory matching the name under the *controllers* folder, in our case the api/` directory.

Rails media types supported

Rails can handle up to 35 different media types, you can list them by accessing the SET class under de Mime module:

```
$ rails c
2.6.3 :001 > Mime::SET.collect(&:to_s)
 => ["text/html", "text/plain", "text/javascript",
"text/css", "text/calendar", "text/csv", "text/vcard",
"text/vtt", "image/png", "image/jpeg", "image/gif",
"image/bmp", "image/tiff", "image/svg+xml", "video/mpeg",
"audio/mpeg", "audio/ogg", "audio/aac", "video/webm",
"video/mp4", "font/otf", "font/ttf", "font/woff",
"font/woff2", "application/xml", "application/rss+xml",
"application/atom+xml", "application/x-yaml",
"multipart/form-data", "application/x-www-form-
urlencoded", "application/json", "application/pdf",
"application/zip", "application/gzip"]
```

This is important because we are going to be working with JSON, one of the built-in MIME types accepted by Rails, so we just need to specify this format as the default one:

config/routes.rb

```
Rails.application.routes.draw do
  # API definition
  namespace :api, defaults: { format: :json }  do
    # We are going to list our resources here
  end
end
```

Up to this point, we have not made anything crazy. What we want to generate is a *base_uri* which includes the API version. But let's commit changes before going to the next section:

```
$ git add config/routes.rb
$ git commit -m "Set the routes constraints for the API"
```

Api versioning

At this point, we should have a nice route mapping using a namespace. Your routes.rb file should look like this:

config/routes.rb

```
Rails.application.routes.draw do
  # API definition
  namespace :api, defaults: { format: :json }  do
    # We are going to list our resources here
  end
end
```

Now it is time to set up some other constraints for versioning purposes. You should care about versioning your application from the beginning since this will give a better structure to your API. When changes need to be made, you can give developers who are consuming your API the opportunity to adapt to the new features while the old ones are being deprecated. There is an excellent railscast explaining this.

To set the version for the API, we first need to add another directory under the API we created:

```
$ mkdir app/controllers/API/v1
```

This way we can namespace our api into different versions very easily, now we just need to add the necessary code to the routes.rb file:

config/routes.rb

```
Rails.application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json }  do
    namespace :v1 do
      # We are going to list our resources here
    end
  end
end
```

By this point, the API is now scoped via the URL. For example, with the current configuration, an endpoint for retrieving a product would

be like http://localhost:3000/v1/products/1.

Common API patterns

You can find many approaches to set up the *base_uri* when building an API following different patterns, assuming we are versioning our API:

- api.example.com/: In my opinion, this is the way to go, gives you a better interface and isolation, and in the long term can help you to quickly scalate
- example.com/api/: This pattern is very common, and it is actually a good way to go when you don't want to namespace your API under a subdomain
- example.com/api/v1: it seems like a good idea by setting the version of the API through the URL seems like a more descriptive pattern, but this way you enforce the version to be included on URL on each request, so if you ever decide to change this pattern, this becomes a problem of maintenance in the long-term

There are some practices in API building that recommend not to version the API via the URL. That's true. The developer should not be aware of the version he's using. For simplicity, I have chosen to set aside this convention, which we will be able to apply in a second phase.

It is time to *commit*:

```
$ git commit -am "Set the versioning namespaces for API"
```

We are at the end of our chapter. Therefore, it is time to apply all our modifications to the master branch by making a *merge*. To do this, we place ourselves on the master branch and we *merge* chapter02:

```
$ git checkout master
$ git merge chapter02
```

Conclusion

I know it’s been a long way, but you made it, don’t give up this is just our small scaffolding for something big, so keep it up. In the meantime, and if you feel curious, some gems handle this kind of configuration:

- RocketPants
- Versionist

I’m not covering those in this book since we are trying to learn how to implement this kind of functionality, but it is good to know. By the way, the code up to this point is here.