{"APIonRails":5}

{"author": "Alexandre Rousseau"}

API on Rails 5

Alexandre Rousseau

Version 6.0.4, 2019-11-08

Table of Contents

Before

Foreword

"API on Rails 5" 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.

Since the original work was not maintained, I wanted to update this excellent work and contribute to the Francophone community by translating it myself. This update is also available in the Molière language [1].

[1] It means french.

About the author

Abraham Kuri is a Rails developer with 5 years of experience (probably more now). His experience includes working as a freelancer in software product development and more recently in collaboration within the open source community. A graduate in computer science from ITESM, he founded two companies in Mexico (Icalia Labs and Codeando Mexico).

On my side, 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) where I build and maintain a SAAS product using Rails. I also contribute to the Ruby community by producing and maintain some gems that you can consult onhttps://rubygems.org/profiles/madeindjs[my Rubygems.org profile]. Most of my projects are on GitHub so don't hesitate to follow me.

All the source code of this book is available in Asciidoctor format on GitHub. So don't hesitate to forke the project if you want to improve it or fix a mistake that 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

Thanks

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

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

Introduction

Welcome to APIs on Rails a tutorial on steroids on how to build your next API with Rails. The goal of this book is to provide an answer on how to develop a RESTful API following the best practices out there, along with my own experience. By the time you are done with *API's on Rails* you should be able to build your own API and integrate it with any clients such as a web browser or your next mobile app. The code generated is built on top of Rails 4 which is the current version, for more information about this check out http://rubyonrails.org/. The most up-to-date version of the *API's on Rails* can be found on APIs on Rails; don't forget to update your offline version if that is the case.

The intention with this book it's not only to teach you how to build an API with Rails. The purpose is also to teach you how to build scalable and maintainable API with Rails which means improve your current Rails knowledge. In this journey we are going to take, you will learn to:

- Build JSON responses
- Use Git for version controlling
- Testing your endpoints
- Optimize and cache the API

I highly recommend you go step by step on this book, try not to skip chapters, as I mention tips and interesting facts for improving your skills on each on them. You can think yourself as the main character of a video game and with each chapter you'll get a higher level.

In this first chapter I will walk you through on how to setup your environment in case you don't have it already. We'll then create the application called market_place_api. I'll emphasize all my effort into teaching you all the best practices I've learned along the years, so this means right after initializing the project we will start tracking it with Git.

In the next chapters we will be building the application to demonstrate a simple workflow I use on my daily basis. We'll develop the whole application using **test driven development** (TDD), getting started by explaining why you want to build an API's for your next project and deciding whether to use JSON or XML as the response format. We'll get our hands dirty then and complete the foundation for the application by building all the necessary endpoints, securing the API access and handling authentication through headers exchange.

Finally on the last chapter we'll add some optimization techniques for improving the server responses.

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.

By the end or during the process (it really depends on your expertise), you will get better and be able to better understand some of the bests Rails resources out there. I also took some of the practices from these guys and brought them to you:

- Railscasts
- CodeSchool
- JSON API

Conventions on this book

The conventions on 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 commands. I won’t deal with windows `cmd` (sorry guys), so I’ll based all the examples using 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’ll will recommend you to `google it', which I don’t consider a bad practice or whatsoever. But if you feel like want to grab a beer or have troubles 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 to 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 comes 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 is to use 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 will 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 probable have a nice terminal already, but the default should work just 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 nice inspector not just for the DOM but 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 less troubles when installing software and managing 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 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 5 requires you to install version 2.2.2 or higher. I recommend you to start using Ruby Version Manager (RVM) or rbenv, any of these will allow you to install multiple versions of `ruby`. We will use RVM in this tutorial any of these two options you choose is fine.

To install RVM go on https://rvm.io/ and get GPG key [2]. Then:

```
$ 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 2.5
```

If everything went smooth, 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
```

On 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:

```
$ printf 'gem: --no-document' >> ~/.gemrc
$ gem install bundler
$ gem install foreman
$ gem install rails -v 5.2
```

Check for everything to be running nice and smooth:

```
$ rails -v 5.2
5.2.0
```

Database

I highly recommend you install Postgresql to manage your databases, but for simplicity we’ll be using SQlite. 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
```

[2] The GPG allow you to verify author identity of the software you download.

Initializing the project

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

Be aware that we'll be using Rspec as the testing suite. So we will use the `--skip-test` option. Also we will use `--api` option.

NOTE	This option came with Rails 5 and it allow to limit gems and Middleware. It will also avoid to generate HTML views when using Rails generators.

There is the command:

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

As you may guess, the commands above will generate the bare bones of your Rails application. The next step is to add some `gems` we'll be using to build the api.

Installing Pow or Prax

You may ask yourself

> Why in the hell would I want to install this type of package?

and the answer is simple, we will be working with subdomains, and in this case using services like Pow or Prax help us achieve that very easily

Installing Pow

NOTE	Pow only works on Mac OS, but don't worry there is an alternative which mimics the functionality on Linux.

To install it just type in:

```
$ curl get.pow.cx | sh
```

And that's it you are all set. You just have to symlink the application in order to set up the Rack app. First you go the `~/.pow` directory:

```
$ cd ~/.pow
```

Then you create the symlink:

```
$ ln -s ~/workspace/market_place_api
```

Remember to change the user directory to the one matches yours. You can now access the application through http://market_place_api.dev/. Your application should be up a running by now.

Installing Prax

For linux users only, Prax distribute some Debian/Ubuntu precompiled packages. You only have to download .deb and instal with dpkg.

```
$ cd /tmp
$ wget
https://github.com/ysbaddaden/prax.cr/releases/download/v0.8.0/p
rax_0.8.0-1_amd64.deb
$ sudo dpkg -i prax_0.8.0-1_amd64.deb
```

Then we just need to link the apps:

```
$ cd ~/workspace/market_place_api
$ prax link
```

If you want to start the Prax server automatically, add this line to the .profile file:

```
prax start
```

NOTE When using prax, you have to specify the port for the URL, in this case http://market_place_api.dev:3000

You should see the application up and running, see image bellow:

Gemfile and Bundler

Once the Rails application is created, the next step is adding a simple but very powerful gem to serialize the resources we are going to expose on the api. The gem is called active_model_serializers which is an excellent choice to go when building this type of application. It is well maintained and the documentation is amazing.

So your Gemfile should look like this after adding the active_model_serializers gem:

Gemfile

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.5.3'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.2.0'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use Puma as the app server
gem 'puma', '~> 3.11'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Api gems
gem 'active_model_serializers'
# ...
```

NOTE I remove the jbuilder and turbolinks gems because we are not really going to use them anyway.

It is a good practice also to include the ruby version used on the whole project, this prevents dependencies to break if the code is shared among different developers, whether if is a private or public project.

It is also important that you update the Gemfile to group the different gems into the correct environment

Gemfile

```
# ...
group :development do
  gem 'sqlite3'
end
# ...
```

This as you may recall will prevent sqlite from being installed or required when you deploy your application to a server provider like Heroku.

NOTE Due to the structure of the application we are not going to deploy the app to any server, but we will be using Pow by Basecamp. If you are using Linux there is a similar solution called Prax by ysbaddaden

Pow is a zero-config Rack server for Mac OS X. Have it serving your apps locally in under a minute. - Basecamp

Once you have this configuration set up, it is time to run the bundle install command to integrate the corresponding dependencies:

```
$ bundle install
```

After the command finish its execution, it is time to start tracking the project with Git.

Versioning

Remember that Git helps you track and maintain history of your code. Keep in mind source code of the application is published on GitHub. You can follow the repository at GitHub. I'll assume you have Git already configured and ready to use to start tracking the project. If that is not your case, follow these first-time setup steps:

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

NOTE Replace the last command editor(`"mvim -f"`) with the one you installed `"subl -w"` for SublimeText ,`"mate -w"` for TextMate, or `"gvim -f"` for gVim.

So it is now time to **init** the project with git. Remember to navigate to the root directory of the `market_place_api` application:

```
$ git init
Initialized empty Git repository in
~/workspace/market_place_api/.git/
```

The next step is to ignore some files that we don't want to track, so your `.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/*

/node_modules
/yarn-error.log

/public/assets
.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 commands necessary are shown below:

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

TIP I have encounter that committing with a message starting with a present tense verb, describes what the commit does and not what it did, this way when you are exploring the history of the project it is more natural to read and understand(or at least for me). 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 in 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.git
```

Then:

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

As we move forward with the tutorial, I’ll be using the practices I follow on my daily basis, 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 from this point things will get better. 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 the bare bones of the application. If you did not read it I recommend you to do it.

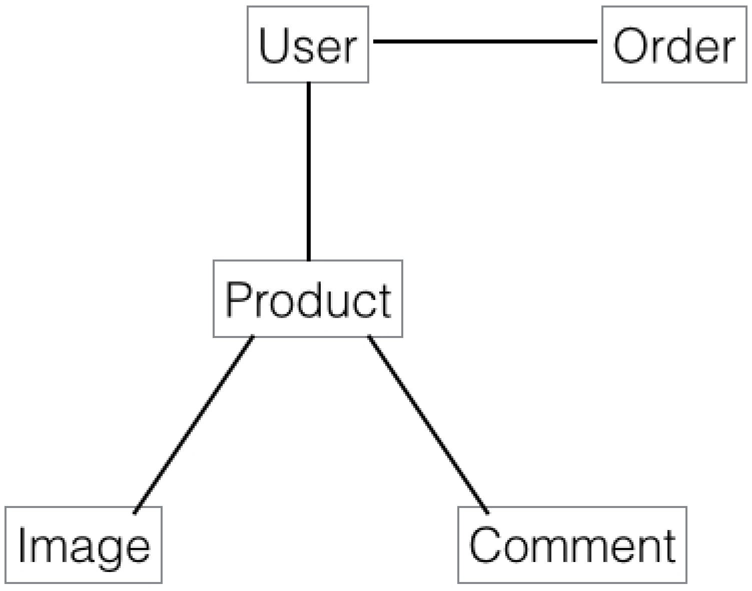
You can clone the project until this point with:

```
$ git clone https://github.com/madeindjs/market_place_api
$ cd market_place_api
$ git checkout -b chapter1
b98a9a7a328017640482af95beebc1d6e612e0ac
```

And as a quick recap, we really just update the `Gemfile` to add the `active_model_serializers` gem.

Planning the application

As we want to go simple with the application, it consists on 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 short terms we have the user who will be able to place many orders, upload multiple products which can have many images or comments from another users on the app.

We are not going to 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, from javascript frameworks(Angular, EmberJS, Backbone) to mobile consumption(AFNetworking).

By this point you must be asking yourself, all right but I need to explore or visualize the api we are going to be building, and that's fair. Probably if you google something related to api exploring, an application called Postman will pop. It is a great software but we won't be using that anyway because we'll use cURL who allow anybody to reproduce request 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 in here.

JSON as the Internet media type is highly accepted because of readability, extensibility and easy to implement in fact many of the current frameworks consume JSON api's by default, in JavaScript there is Angular or EmberJS, but there are great libraries for objective-c too like AFNetworking or RESTKit. There are probably good solutions for Android, but because of my lack of experience on that development platform I might not be the right person to recommend you something.

All right, so we are building our API with JSON, but there are many ways to achieve this, the first thing that could come to your mind would be just to start dropping some routes defining the end points but 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

> is a way to create, read, update or delete information on a server using simple HTTP calls. It is an alternative to more complex mechanisms like SOAP, CORBA and RPC. A REST call is simply a GET HTTP request to the server.

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

And in REST you may call a URL with an e 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, it is commonly JSON and is commonly set through headers exchange.
- Follow 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 as we move on with the tutorial, hopefully it'll get a lot easier to understand.

Routes, Constraints and Namespaces

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

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

We are only going to be working on the `config/routes.rb`, as we are just going to set the `constraints`, the `base_uri` 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, in Rails this is fairly simple, just create a folder under the `app/controllers` named `api`, the name is important as it is the namespace we'll use for managing the controllers for the api endpoints.

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

We then add that namespace into our routes.rb file:

config/routes.rb

```ruby
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
Loading development environment (Rails 5.2.1)
irb(main):001:0> 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",
"application/vnd.web-console.v2"]
```

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 to generate a *base_uri* under a subdomain. In our case something like `api.market_place_api.dev`. Setting the api under a subdomain is a good practice because it allows to scale the application to a DNS level. So how do we achieve that?

config/routes.rb

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

Notice the changes? We didn’t just add an `Hash` constraints to specify the subdomain. We also add the `path` option, and set it on root path (`/`). This is telling Rails to set the starting path for each request to be root in relation to the subdomain, achieving what we are looking for.

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/`: I 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`: his 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

Don't worry about versioning right now, I'll walk through it later.

Time to commit:

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

All right take a deep breath, drink some water, and let's get going.

Api versioning

At this point we should have a nice routes mapping using a subdomain for name spacing the requests, your `routes.rb` file should look like this:

config/routes.rb

```
Rails.application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' }  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, and when changes need to be done, you can give developers who are consuming your api the opportunity to adapt for the new features while the old ones are being deprecated. There is an excellent railscast explaining this.

In order 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 scope 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 }, constraints: {
subdomain: 'api' }  do
    scope module: :v1 do
      # We are going to list our resources here
    end
  end
end
```

By this point the API is now scoped via de URL. For example with the current configuration an end point for retrieving a product would be like: http://api.marketplace.dev/v1/products/1.

Improving the versioning

So far we have the API versioned scoped via the URL, but something doesn't feel quite right, isn't it?. What I mean by this is that from my point of view the developer should not be aware of the version using it, as by default they should be using the last version of your endpoints, but how do we accomplish this?.

Well first of all, we need to improve the API version access through HTTP Headers. This has two benefits:

- Removes the version from the URL
- The API description is handle through request headers

Most commons HTTP headers fields

HTTP header fields are components of the message header of requests and responses in the Hypertext Transfer Protocol (HTTP). They define an operating parameters of an HTTP transaction. A common list of used headers is presented below:

- **Accept**: Content-Types that are acceptable for the response. Example: `Accept: text/plain`
- **Authorization**: Authentication credentials for HTTP authentication. Example: `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`
- **Content-Type**: The MIME type of the body of the request (used with POST and PUT requests). Example: `Content-Type: application/x-www-form-urlencoded`
- **Origin**: Initiates a request for cross-origin resource sharing (asks server for an `Access-Control-Allow-Origin' response header). Example: `Origin: http://www.example-social-network.com
- **User-Agent**: The user agent string of the user agent. Example: `User-Agent: Mozilla/5.0`

It is important that you feel comfortable with this ones and understand them.

In Rails is very easy to add this type versioning through an *Accept* header. We will create a class under the `lib` directory of your rails app, and remember we are doing TDD so first things first.

First we need to add our testing suite, which in our case is going to be Rspe:

Gemfile

```
group :test do
  gem 'rspec-rails', '~> 3.8'
  gem 'factory_bot_rails', '~> 4.9'
  gem 'ffaker', '~> 2.10'
end
```

Then we run the bundle command to install the gems

```
$ bundle install
```

Finally we install the `rspec` and add some configuration to prevent views and helpers tests from being generated:

```
$ rails generate rspec:install
```

config/application.rb

```ruby
# ...
module MarketPlaceApi
  class Application < Rails::Application
    # Initialize configuration defaults for originally generated
Rails version.
    config.load_defaults 5.2

    config.generators do |g|
      g.test_framework :rspec, fixture: true
      g.fixture_replacement :factory_bot, dir: 'spec/factories'
      g.view_specs false
      g.helper_specs false
      g.stylesheets = false
      g.javascripts = false
      g.helper = false
    end

    config.autoload_paths += %W(\#{config.root}/lib)

    # Don't generate system test files.
    config.generators.system_tests = nil
  end
end
```

If everything went well it is now time to add a `spec` directory under `lib` and add the `api_constraints_spec.rb`:

```
$ mkdir lib/spec
$ touch lib/spec/api_constraints_spec.rb
```

We then add a bunch of specs describing our class:

lib/spec/api_constraints_spec.rb

```
require 'spec_helper'
require './lib/api_constraints'

describe ApiConstraints do
  let(:api_constraints_v1) { ApiConstraints.new(version: 1) }
  let(:api_constraints_v2) { ApiConstraints.new(version: 2,
default: true) }

  describe 'matches?' do
    it "returns true when the version matches the 'Accept'
header" do
      request = double(host: 'api.marketplace.dev',
                       headers: { 'Accept' =>
'application/vnd.marketplace.v1' })
      expect(api_constraints_v1.matches?(request)).to be_truthy
    end

    it "returns the default version when 'default' option is
specified" do
      request = double(host: 'api.marketplace.dev')
      expect(api_constraints_v2.matches?(request)).to be_truthy
    end
  end
end
```

Let me walk you through the code. We are initializing the class with an `Hash` option. `Hash` option will contain the version of the API and a default value for handling the default version. We provide a `matches?` method which the router will trigger for the constraint to see if the default version is required or the `Accept` header matches the given string.

The implementation looks likes this

lib/api_constraints.rb

```
class ApiConstraints
  def initialize(options)
    @version = options[:version]
    @default = options[:default]
  end

  def matches?(req)
    @default || req.headers['Accept'].include?
("application/vnd.marketplace.v#{@version}")
  end
end
```

As you imagine we need to add the class to our `routes.rb` file and set it as a constraint scope option.

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      # We are going to list our resources here
    end
  end
end
```

The configuration above now handles versioning through headers, and for now the version 1 is the default one, so every request will be redirected to that version, no matter if the header with the version is present or not.

Before we say goodbye, let’s run our first tests and make sure everything is nice and green:

```
$ bundle exec rspec lib/spec/api_constraints_spec.rb
..

Finished in 0.00294 seconds (files took 0.06292 seconds to load)
2 examples, 0 failures
```

Conclusion

It’s been a long way, I know, 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 I you feel curious there are some gems that handle this kind of configuration:

- RocketPants
- Versionist

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

Presenting the users

In the last chapter we manage to set up the bare bones for our application endpoints configuration. We even added versioning through headers. In a next chapter we will handle users authentication through authentication tokens as well as setting permissions to limit access for let's say signed in users. In coming chapters we will relate products to users and give them the ability to place orders.

You can clone the project until this point with:

```
$ git clone
https://github.com/madeindjs/market_place_api/tree/chapitre_2
```

As you can already imagine there are a lot of authentication solutions for Rails, AuthLogic, Clearance and Devise. We will be using the last one, which offers a great way to integrate not just basic authentication, but many other modules for further use.

Devise

Devise comes with up to 10 modules for handling authentication

- Database Authenticable
- Omniauthable
- Confirmable
- Recoverable
- Registerable
- Rememberable
- Trackable
- Timeoutable
- Validatable
- Lockable

If you have not work with devise before, I recommend you visit the reposity page and read the documentation, there are a lot of good examples out there too.

This is going to be a full-packed chapter. It may be long but I'm trying to cover as many topics along with best practices on the way, so feel free to grab a cup of coffee and let’s get going. By the end of the chapter you will have full user endpoints, along with validations and error server responses.

We want to track this chapter, so it would be a good time to create a new branch for this:

```
$ git checkout -b chapter3
```

Just make sure you are on the `master` branch before checking out.

User model

We need to first add the devise gem into the Gemfile

Gemfile

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.5.3'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 5.2.0'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use Puma as the app server
gem 'puma', '~> 3.11'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Api gems
gem 'active_model_serializers'

# Reduces boot times through caching; required in config/boot.rb
gem 'bootsnap', '>= 1.1.0', require: false

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get
a debugger console
  gem 'byebug', platforms: %i[mri mingw x64_mingw]
end

group :development do
  # Access an interactive console on exception pages or by
calling 'console' anywhere in the code.
  gem 'listen', '>= 3.0.5', '< 3.2'
  gem 'web-console', '>= 3.3.0'
  # Spring speeds up development by keeping your application
running in the background. Read more:
https://github.com/rails/spring
  gem 'spring'
  gem 'spring-watcher-listen', '~> 2.0.0'
end
```

```
group :test do
  gem 'factory_bot_rails', '~> 4.9'
  gem 'ffaker', '~> 2.10'
  gem 'rspec-rails', '~> 3.8'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-
data gem
gem 'tzinfo-data', platforms: %i[mingw mswin x64_mingw
jruby]

gem 'devise'
```

Then run the bundle install command to install it. Once the bundle command finishes, we need to run the devise install generator:

```
$ rails g devise:install
  create  config/initializers/devise.rb
  create  config/locales/devise.en.yml
  ...
```

By now and if everything went well we will be able to generate the user model through the devise generator:

```
$ rails g devise User
    invoke  active_record
    create    db/migrate/20181113070805_devise_create_users.rb
    create    app/models/user.rb
    invoke    rspec
    create      spec/models/user_spec.rb
    invoke      factory_bot
    create        spec/factories/users.rb
    insert    app/models/user.rb
    route  devise_for :users
```

From now every time we create a model, the generator will also create a factory file for that model. This will help us to easily create test users and facilitate our tests writing.

spec/factories/users.rb

```
FactoryBot.define do
  factory :user do

  end
end
```

Next we migrate the database and prepare the test database.

```
$ rake db:migrate
== 20181113070805 DeviseCreateUsers: migrating
================================
-- create_table(:users)
   -> 0.0008s
-- add_index(:users, :email, {:unique=>true})
   -> 0.0005s
-- add_index(:users, :reset_password_token, {:unique=>true})
   -> 0.0007s
== 20181113070805 DeviseCreateUsers: migrated (0.0023s)
=======================
```

```
$ rake db:test:prepare
```

Let’s commit this, just to keep our history points very atomic.

```
$ git add .
$ git commit -m "Adds devise user model"
```

First user tests

We will add some specs to make sure the user model responds to the email, password and password_confirmation attributes provided by devise, let's add them. Also for convenience we will modify the users factory file to add the corresponding attributes.

spec/factories/users.rb

```ruby
FactoryBot.define do
  factory :user do
    email { FFaker::Internet.email }
    password { '12345678' }
    password_confirmation { '12345678' }
  end
end
```

Once we'd added the attributes it is time to test our User model.

spec/models/user_spec.rb

```ruby
# ...
RSpec.describe User, type: :model do
  before { @user = FactoryBot.build(:user) }
  subject { @user }
  it { should respond_to(:email) }
  it { should respond_to(:password) }
  it { should respond_to(:password_confirmation) }
  it { should be_valid }
end
```

Because we previously prepare the test database, with rake db:test:prepare, we just simply run the tests:

```
$ bundle exec rspec spec/models/user_spec.rb
....

Finished in 0.03231 seconds (files took 0.81624 seconds to load)
4 examples, 0 failures
```

That was easy, we should probably commit this changes:

```
$ git add .
$ git commit -am 'Adds user firsts specs'
```

Improving validation tests

It is showtime people, we are building our first endpoint. We are just going to start building the `show` action for the user which is going to expose a `user` record in plain old JSON. We first need to generate the `users_controller`, add the corresponding tests and then build the actual code.

First we generate the `users` controller:

```
$ rails generate controller users
```

This command will create a `users_controller_spec.rb`. Before we get into that, there are 2 basic steps we should be expecting when testing `api` endpoints.

- The JSON structure to be returned from the server
- The status code we are expecting to receive from the server

Most common http codes

The first digit of the status code specifies one of five classes of response; the bare minimum for an HTTP client is that it recognize these five classes. A common list of used http codes is presented below:

- 200: Standard response for successful HTTP requests (It is commonly on GET requests)
- 201: The request has been fulfilled and resulted in a new resource being created (After POST requests)
- 204: The server successfully processed the request, but is not returning any content (It is usually a successful DELETE request)
- 400: The request cannot be fulfilled due to bad syntax.
- 401: Similar to 403 Forbidden, but specifically for use when authentication is required and has failed or has not yet been provided
- 404: The requested resource could not be found but may be available again in the future (Usually GET requests)
- 500: A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

For a full list of HTTP method check out the article on Wikipedia talking about it

To keep our code nicely organized, we will create some directories under the controller specs directory in order to be consistent with our current setup. There is also another set up out there which uses instead of the `controllers` directory a `request` or `integration` directory, I this case I like to be consistent with the `app/controllers` directory.

```
$ mkdir -p spec/controllers/api/v1
$ mv spec/controllers/users_controller_spec.rb
spec/controllers/api/v1
```

After creating the corresponding directories we need to change the file `describe` name from `UsersController` to `Api::V1::UsersController`, the updated file should look like:

spec/controllers/api/v1/users_controller_spec.rb

```ruby
RSpec.describe Api::V1::UsersController, type: :controller do

end
```

Now with tests added your file should look like:

spec/controllers/api/v1/users_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  before(:each) { request.headers['Accept'] =
"application/vnd.marketplace.v1" }

    describe "GET #show" do
      before(:each) do
        @user = FactoryBot.create :user
        get :show, params: { id: @user.id, format: :json}
      end

      it "returns the information about a reporter on a hash" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:email]).to eql @user.email
      end

      it { expect(response).to be_success }
    end
end
```

So far, the tests look good, we just need to add the implementation. It is extremely simple:

app/controllers/api/v1/users_controller.rb

```ruby
class  Api::V1::UsersController < ApplicationController
  def show
    render json: User.find(params[:id])
  end
end
```

You may activate `Devise::Test::ControllerHelpers` module in `spec/rails_helper.rb` file to load helpers. To do so you only have to add this line:

```
# ...
RSpec.configure do |config|
  # ...
  config.include Devise::Test::ControllerHelpers, type:
:controller
  # ...
end
```

If you run the tests now with `rspec spec/controllers` you will see an error message similar to this:

```
$ rspec spec/controllers
FF

Failures:

  1) Api::V1::UsersController GET #show returns the information
about a reporter on a hash
    Failure/Error: get :show, params: { id: @user.id, format:
:json}

    ActionController::UrlGenerationError:
    No route matches {:action=>"show",
:controller=>"api/v1/users", :format=>:json, :id=>1}
      ...

  2) Api::V1::UsersController GET #show
    Failure/Error: get :show, params: { id: @user.id, format:
:json}


    ActionController::UrlGenerationError:
    No route matches {:action=>"show",
:controller=>"api/v1/users", :format=>:json, :id=>1}
      ...

Finished in 0.01632 seconds (files took 0.47675 seconds to load)
  2 examples, 2 failures
```

This kind of error if very common when generating endpoints manually, we totally forgot the `routes`. So let's add them:

config/routes.rb

```
# ...
Rails.application.routes.draw do
  devise_for :users
  # Api definition
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      resources :users, only: [:show]
    end
  end
end
```

Tests should now pass:

```
$ bundle exec rspec spec/controllers
..

Finished in 0.02652 seconds (files took 0.47291 seconds to load)
2 examples, 0 failures
```

As usual and after adding some bunch of code we are satisfied with, we commit the changes:

```
$ git add .
$ git commit -m "Adds show action the users controller"
```

Testing endpoints with CURL

So we finally have an endpoint to test. There are plenty of options to start playing with. The first that come to my mind is using cURL because is built-in on almost any Linux distribution and of course on your Mac OSX. So let’s try it out:

NOTE Remember our base URI is api.market_place_api.dev.

```
$ curl -H 'Accept: application/vnd.marketplace.v1'
http://api.market_place_api.dev/users/1
```

This will throw us an error. Well you might expect that already because we don't have a user with `id` equals to 1. Let's create it first through the terminal:

```
$ rails console
Loading development environment (Rails 5.2.1)
2.5.3 :001 >  User.create email: "example@marketplace.com",
password: "12345678", password_confirmation: "12345678"
```

After creating the user successfully our endpoint should work:

```
$ curl -H 'Accept: application/vnd.marketplace.v1' \
http://api.market_place_api.dev/users/1
{"id":1,"email":"example@marketplace.com", ...
```

So there you go. You now have a user record API endpoint. If you are having problems with the response and double checked everything is well assembled. Well then you might need to visit the `application_controller.rb` file and update it a little bit like so

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::API
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :null_session
end
```

As suggested even by Rails we should be using `null_session` to prevent CSFR attacks from being raised, so **I highly recommend you do it as this will not allow POST or PUT requests to work**. After updating the `application_controller.rb` file it is probably a good point to place a commit:

```
$ git add .
$ git commit -m "Updates application controller to prevent CSRF
exception from being raised"
```

Creating users

Now that we have a better understanding on how to build endpoints and how they work, it's time to add more abilities to the API. One of the most important is letting the users actually create a profile on our application. As usual we will write tests before implementing our code extending our testing suite.

Creating records in Rails as you may know is really easy, the trick when building an api is which is the best fit for the HTTP codes to send on the response, as well as the actual `json response`. If you don't totally get this it will probably be more easy on the code:

Make sure your repository is clean and that you don't have any commits left, if so place them so we can start fresh.

Let's proceed with our test-driven development by adding a `create` endpoint on the `users_controller_spec.rb` file

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  # ...
  describe 'POST #create' do
    context 'when is successfully created' do
      before(:each) do
        @user_attributes = FactoryBot.attributes_for :user
        post :create, params: { user: @user_attributes },
format: :json
      end

      it 'renders the json representation for the user record
just created' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:email]).to eql @user_attributes
[:email]
      end

      it { expect(response.response_code).to eq(201) }
    end

    context 'when is not created' do
      before(:each) do
        # notice I'm not including the email
        @invalid_user_attributes = { password: '12345678',
                                     password_confirmation:
'12345678' }
        post :create, params: { user: @invalid_user_attributes
}, format: :json
      end

      it 'renders an errors json' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response).to have_key(:errors)
      end

      it 'renders the json errors on why the user could not be
created' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:errors][:email]).to include "can't
be blank"
      end

      it {  expect(response.response_code).to eq(422) }
    end
  end
end
```

There is a lot of code up there but don't worry I'll walk you through it:

- We need to validate to states on which the record can be, valid or invalid. In this case we are using the context clause to achieve this scenarios.
- In case everything goes smooth, we should return a 201 HTTP code which means a record just got created, as well as the JSON representation of that object.
- In case of any errors, we have to return a 422 HTTP code which stands for Unprocessable Entity meaning the server could save the record. We also return a JSON representation of why the resource could not be saved.

If we run our tests now, they should fail:

```
$ rspec spec/controllers/api/v1/users_controller_spec.rb
.FFFFFF
```

Time to implement some code and make our tests pass:

app/controllers/api/v1/users_controller.rb

```ruby
class Api::V1::UsersController < ApplicationController
  # ...
  def create
    user = User.new user_params
    if user.save
      render json: user, status: 201, location: [:api, user]
    else
      render json: { errors: user.errors }, status: 422
    end
  end

  private

  def user_params
    params.require(:user).permit(:email, :password,
:password_confirmation)
  end
end
```

Remember that each time we add an endpoint we have to add that action into our `routes.rb` file

config/routes.rb

```ruby
Rails.application.routes.draw do
  # ...
  resources :users, only: [:show, :create]
  # ...
end
```

As you can see the implementation is fairly simple. We also added the `user_params` private method to sanitize the attribute to be assigned through mass-assignment. Now if we run our tests, they all should be nice and green:

```
$ bundle exec rspec
spec/controllers/api/v1/users_controller_spec.rb
.......

Finished in 0.05967 seconds (files took 0.4673 seconds to load)
7 examples, 0 failures
```

Let’s commit the changes and continue building our application:

```
$ git add .
$ git commit -m "Adds the user create endpoint"
```

Update users

The pattern for **updating** users is very similar as **creating** new ones. If you are an experienced Rails developer you may already know the differences between these two actions:

- The `update` action responds to a PUT/PATCH request.
- Only the `current_user` should be able to update their information, meaning we have to enforce a user to be authenticated. We will cover that on next chapters

As usual we start by writing our tests:

spec/controllers/api/v1/users_controller_spec.rb

```
RSpec.describe Api::V1::UsersController, type: :controller do
  # ...
  describe "PUT/PATCH #update" do

    context "when is successfully updated" do
      before(:each) do
        @user = FactoryBot.create :user
        patch :update, params: {
          id: @user.id,
          user: { email: "newmail@example.com" } },
          format: :json
      end

      it "renders the json representation for the updated user" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:email]).to eql
"newmail@example.com"
      end

      it {  expect(response.response_code).to eq(200) }
    end

    context "when is not created" do
```

```
      before(:each) do
        @user = FactoryBot.create :user
        patch :update, params: {
          id: @user.id,
          user: { email: "bademail.com" } },
          format: :json
      end

      it "renders an errors json" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response).to have_key(:errors)
      end

      it "renders the json errors on why the user could not be
created" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:errors][:email]).to include "is
invalid"
      end

      it {  expect(response.response_code).to eq(422) }
    end
  end
end
```

Getting the tests to pass requires us to build the update action on the users_controller.rb file as well as adding it to the routes.rb. As you can see we have to much code duplicated, we'll refactor our tests in next chapter.

First we add the action the routes.rb file

config/routes.rb

```
Rails.application.routes.draw do
  # ...
  resources :users, only: [:show, :create, :update]
  # ...
end
```

Then we implement the update action on the users controller and make our tests pass:

app/controllers/api/v1/users_controller.rb

```
class Api::V1::UsersController < ApplicationController
  # ...
  def update
    user = User.find(params[:id])

    if user.update(user_params)
      render json: user, status: 200, location: [:api, user]
    else
      render json: { errors: user.errors }, status: 422
    end
  end
  # ...
end
```

If we run our tests, we should now have all of our tests passing.

```
$ bundle exec rspec
spec/controllers/api/v1/users_controller_spec.rb
............

Finished in 0.08826 seconds (files took 0.47286 seconds to load)
12 examples, 0 failures
```

We commit the changes as we added a bunch of working code:

```
$ git add .
$ git commit -m "Adds update action the users controller"
```

Destroying users

So far we have built a bunch of actions on the users controller along with their tests but we have not ended yet. We are just missing one more which is the destroy action. So let’s do that:

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  before(:each) { request.headers['Accept'] =
'application/vnd.marketplace.v1' }
  # ...
  describe "DELETE #destroy" do
    before(:each) do
      @user = FactoryBot.create :user
      delete :destroy, params: { id: @user.id }, format: :json
    end

    it { expect(response.response_code).to eq(204) }
  end
end
```

As you can see the spec is very simple. We only respond with a status of `204` which stands for `No Content`. This means that the server successfully processed the request but is not returning any content. We could also return a `200` status code but I find more natural to respond with `No Content` in this case as we are deleting a resource and a success response may be enough.

The implementation for the destroy action is fairly simple as well:

app/controllers/api/v1/users_controller.rb

```
class Api::V1::UsersController < ApplicationController
  # ...
  def destroy
    user = User.find(params[:id])
    user.destroy
    head 204
  end
  # ...
end
```

Remember to add the `destroy` action to the user resources on the `routes.rb` file:

config/routes.rb

```ruby
Rails.application.routes.draw do
  # ...
  resources :users, only: [:show, :create, :update, :destroy]
  # ...
end
```

If you run your tests now, they should be all green:

```
$ bundle exec rspec
spec/controllers/api/v1/users_controller_spec.rb
.............

Finished in 0.09255 seconds (files took 0.4618 seconds to load)
13 examples, 0 failures
```

Remember that after making some changes to our code it is a good practice to commit. This habit will keep our history very atomic.

```
$ git add .
$ git commit -m "Adds destroy action to the users controller"
```

Conclusion

Oh you are here!, great job! I know it probably was a long way, but don’t give up you are doing it great. Make sure you are understanding every piece of code, things will get better, in next chapter we will refactor our tests to clean our code a bit and make it easy to extend the test suite more. So stay with me guys!

Refactoring tests

In previous chapter we manage to put together some user resources endpoints. If you skip it (or simple missed it) I highly recommend you take a look at it. It covers the first test specs and an introduction to JSON responses.

You can clone the project until this point with:

```
$ git clone --branch chapitre_3
https://github.com/madeindjs/market_place_api
```

In this chapter we’ll refactor our test specs by adding some helper methods. We’ll also remove the format param sent on every request and do it through headers. Also we hopefully build more consistent and scalable test suite.

So let’ take a look to the users_controller_spec.rb file:

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  before(:each) { request.headers['Accept'] =
'application/vnd.marketplace.v1' }

  describe 'GET #show' do
    before(:each) do
      @user = FactoryBot.create :user
      get :show, params: { id: @user.id, format: :json }
    end

    it 'returns the information about a reporter on a hash' do
      user_response = JSON.parse(response.body, symbolize_names:
true)
      expect(user_response[:email]).to eql @user.email
    end

    it { expect(response.response_code).to eq(200) }
  end

  describe 'POST #create' do
    context 'when is successfully created' do
      before(:each) do
```

```
        @user_attributes = FactoryBot.attributes_for :user
        post :create, params: { user: @user_attributes },
format: :json
      end

      it 'renders the json representation for the user record
just created' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:email]).to eql @user_attributes
[:email]
      end

      it { expect(response.response_code).to eq(201) }
    end

    context 'when is not created' do
      before(:each) do
        # notice I'm not including the email
        @invalid_user_attributes = { password: '12345678',
                                     password_confirmation:
'12345678' }
        post :create, params: { user: @invalid_user_attributes
}, format: :json
      end

      it 'renders an errors json' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response).to have_key(:errors)
      end

      it 'renders the json errors on why the user could not be
created' do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:errors][:email]).to include "can't
be blank"
      end

      it {  expect(response.response_code).to eq(422) }
    end
  end

  describe "PUT/PATCH #update" do
```

```
    context "when is successfully updated" do
      before(:each) do
        @user = FactoryBot.create :user
        patch :update, params: {
          id: @user.id,
          user: { email: "newmail@example.com" } },
          format: :json
      end

      it "renders the json representation for the updated user" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:email]).to eql
"newmail@example.com"
      end

      it {  expect(response.response_code).to eq(200) }
    end

    context "when is not created" do
      before(:each) do
        @user = FactoryBot.create :user
        patch :update, params: {
          id: @user.id,
          user: { email: "bademail.com" } },
          format: :json
      end

      it "renders an errors json" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response).to have_key(:errors)
      end

      it "renders the json errors on why the user could not be
created" do
        user_response = JSON.parse(response.body,
symbolize_names: true)
        expect(user_response[:errors][:email]).to include "is
invalid"
      end

      it {  expect(response.response_code).to eq(422) }
    end
```

```
  end

  describe "DELETE #destroy" do
    before(:each) do
      @user = FactoryBot.create :user
      delete :destroy, params: { id: @user.id }, format: :json
    end

    it { expect(response.response_code).to eq(204) }
  end
end
```

As you can see there is a lot of duplicated code, two big refactors here are:

- The `JSON.parse` method can be encapsulated on a method.
- The `format` param is sent on every request. Although is not a bad practice but it is better if you handle the response type through headers.

So let's add a method for handling the JSON response. If you have been following the tutorial you may know that we are creating a branch for each chapter. So let's do that:

```
$ git checkout -b chapter4
```

Refactoring the JSON response

Back to the refactor, we will add file under the spec/support directory. Currently we don’t have this directory. So let’s add it:

```
$ mkdir spec/support
```

Then we create a request_helpers.rb file under the just created support directory:

```
$ touch spec/support/request_helpers.rb
```

It is time to extract the JSON.parse method into our own support method:

spec/support/request_helpers.rb

```
module Request
  module JsonHelpers
    def json_response
      @json_response ||= JSON.parse(response.body,
symbolize_names: true)
    end
  end
end
```

We scope the method into some modules just to keep our code nice and organized. The next step here is to update the users_controller_spec.rb file to use the method. A quick example is presented below:

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
it 'returns the information about a reporter on a hash' do
  user_response = json_response # c'est cette ligne qui est maj
  expect(user_response[:email]).to eql @user.email
end
# ...
```

Now it is your turn to update the whole file.

After you are done updating the file and if you tried to run your tests you probably encounter a problem. For some reason it is not finding the `json_response` method which is weird because if we take a look at the `spec_helper.rb` file. We can see that is actually loading all files from the `support` directory:

spec/rails_helper.rb

```ruby
# Load all Ruby files placed in spec/support folder
Dir[Rails.root.join('spec', 'support', '**', '*.rb')].each do |f|
  require f
end

RSpec.configure do |config|
  #  ...
  # We also need to include JsonHelpers methods in Rspec tests
  config.include Request::JsonHelpers, :type => :controller
  #  ...
end
```

After that if we run our tests again, everything should be green again. So let’s commit this before adding more code:

```
$ git add .
$ git commit -m "Refactors the json parse method"
```

Refactoring the format param

We want to remove the `format` param sent on every request and instead of that let's handle the response we are expecting through headers. This is extremely easy just by adding one line to our `users_controller_spec.rb` file:

spec/controllers/api/v1/users_controller_spec.rb

```
RSpec.describe Api::V1::UsersController, type: :controller do
  before(:each) { request.headers['Accept'] =
"application/vnd.marketplace.v1, application/json" }
```

By adding this line, you can now remove all the `format` param we were sending on each request and forget about it for the whole application, as long as you include the `Accept` header with the JSON mime type.

Wait we are not over yet! We can add another header to our request that will help us describe the data contained we are expecting from the server to deliver. We can achieve this fairly easy by adding one more line specifying the `Content-Type` header:

spec/controllers/api/v1/users_controller_spec.rb

```
RSpec.describe Api::V1::UsersController, type: :controller do
  before(:each) { request.headers['Accept'] =
"application/vnd.marketplace.v1, application/json" }
  before(:each) { request.headers['Content-Type'] =
'application/json' }
```

And again if we run our tests we can see they are all nice and green:

```
$ bundle exec rspec
spec/controllers/api/v1/users_controller_spec.rb
.............

Finished in 1.44 seconds (files took 0.4734 seconds to load)
13 examples, 0 failures
```

As always this is a good time to commit:

```
$ git commit -am "Factorize format for unit tests"
```

Refactor before actions

I’m very happy with the code we got so far but we can still improve it a little bit. The first thing that comes to my mind is to group the three custom headers being added before each request:

spec/controllers/api/v1/users_controller_spec.rb

```
#...
before(:each) { request.headers['Accept'] =
"application/vnd.marketplace.v1, application/json" }
before(:each) { request.headers['Content-Type'] =
'application/json' }
```

This is good but not good enough because we will have to add this five lines of code for each file. If for some reason we are changing this (let’s say the response type to `xml`) well you do the math. But don’t worry I provide you a solution which will solve all these problems.

First of all we have to extend our `request_helpers.rb` file to include another module, which I named `HeadersHelpers` and which will have the necessary methods to handle these custom headers *spec/support/request_helpers.rb*

```
module Request
  # ...
  module HeadersHelpers
    def api_header(version = 1)
      request.headers['Accept'] =
"application/vnd.marketplace.v#{version}"
    end

    def api_response_format(format ='application/json')
      request.headers['Accept'] = "#{request.headers['Accept']},
#{format}"
      request.headers['Content-Type'] = format
    end

    def include_default_accept_headers
      api_header
      api_response_format
    end
  end
end
```

As you can see I broke the calls into two methods: one for setting the API header and the other one for setting the response format. Also and for convenience I wrote a method `include_default_accept_headers` for calling those two.

And now to call this method before each of our test cases we can add the `before` hook [3] on the *Rspec.configure* block at `spec_helper.rb` file, and make sure we specify the type to `:controller`, as we don't to run this on unit tests.

spec/rails_helper.rb

```
# ...
RSpec.configure do |config|
  # ...
  config.include Request::HeadersHelpers, :type => :controller
  config.before(:each, type: :controller) do
    include_default_accept_headers
  end
  # ...
end
```

After adding this lines we can remove the before hooks on the users_controller_spec.rb file and check that our tests are still passing. You can review a full version of the spec_helper.rb file below:

spec/rails_helper.rb

```
require 'spec_helper'
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
# Prevent database truncation if the environment is production
abort("The Rails environment is running in production mode!") if
Rails.env.production?
require 'rspec/rails'

Dir[Rails.root.join('spec', 'support', '**', '*.rb')].each { |f|
require f }

begin
  ActiveRecord::Migration.maintain_test_schema!
rescue ActiveRecord::PendingMigrationError => e
  puts e.to_s.strip
  exit 1
end

RSpec.configure do |config|
  config.fixture_path = "#{::Rails.root}/spec/fixtures"
  config.use_transactional_fixtures = true

  config.include Devise::Test::ControllerHelpers, type:
:controller
  config.include Request::JsonHelpers, :type => :controller
  config.include Request::HeadersHelpers, :type => :controller
  config.before(:each, type: :controller) do
    include_default_accept_headers
  end

  config.infer_spec_type_from_file_location!
  config.filter_rails_from_backtrace!
end
```

Well now I do feel satisfied with the code, let's commit the changes:

```
$ git commit -am "Refactors test headers for each request"
```

Remember you can review the code up to this point at the GitHub repository.

[3] An hook allow you to run a specific method when a method is called

Conclusion

Nice job on finishing this chapter. Although it was a short one it was a crucial step as this will help us write better and faster tests. On next chapter we will add the authentication mechanism so we’ll be using across the application as well as limiting the access for certain actions.

Authenticating users

It’s been a long way since you started. I hope you are enjoying this trip as much as me. On previous chapter we refactor our test suite and since we did not add much code. If you skipped that chapter I recommend you read it. As we are going to be using some of the methods in the chapters to come.

You can clone the project up to this point:

```
$ git clone --branch chapitre_4
https://github.com/madeindjs/market_place_api
```

In this chapter things will get very interesting because we are going to set up our authentication mechanism. In my opinion this is going to be one of the most interesting chapters. We will introduce a lot of new terms and you will end with a simple but powerful authentication system. Don’t feel panic we will get to that.

First things first (and as usual when starting a new chapter) we will create a new branch:

```
$ git checkout -b chapter5
```

Stateless session

Before we go any further, something must be clear: **an API does not handle sessions**. If you don't have experience building these kind of applications it might sound a little crazy but stay with me. An API should be stateless which means by definition *is one that provides a response after your request, and then requires no further attention.*. Which means no previous or future state is required for the system to work.

The flow for authenticating the user through an API is very simple:

1. The client request for `sessions` resource with the corresponding credentials, usually email and password.
2. The server returns the `user` resource along with its corresponding authentication token
3. Every page that requires authentication, the client has to send that `authentication token`

Of course this is not the only 3-step to follow, and even on step 2 you might think, well do I really need to respond with the entire user or just the `authentication token`, I would say, it really depends on you, but I like to return the entire user, this way I can map it right away on my client and save another possible request from being placed.

In this section and the next we will be focusing on building a Sessions controller along with its corresponding actions. We'll then complete the request flow by adding the necessary authorization access.

Authentication token

Before we proceed with the logic on the sessions controller, we have to first add the `authentication token` field to the `user` model and then add a method to actually set it.

First we generate the migration file:

```
$ rails generate migration add_authentification_token_to_users
auth_token:string
```

As a good practice I like to setup `string` values to an empty string. Let's add an index with a unique truly condition. This way we warranty there are no users with the same token at a database level. So let's do that:

db/migrate/20181114134521_add_authentification_token_to_users.rb

```
class AddauthentificationTokenToUsers < ActiveRecord::Migration
[5.2]
  def change
    add_column :users, :auth_token, :string, default: ''
    add_index :users, :auth_token, unique: true
  end
end
```

Then we run the migrations to add the field and prepare the test database:

```
$ rake db:migrate
== 20181114134521 AddauthentificationTokenToUsers: migrating
====================
-- add_column(:users, :auth_token, :string, {:default=>""})
   -> 0.0004s
-- add_index(:users, :auth_token, {:unique=>true})
   -> 0.0010s
== 20181114134521 AddauthentificationTokenToUsers: migrated
(0.0016s) ===========
```

Now it would be a good time to add some response and uniqueness tests to our `user` model spec:

spec/models/user_spec.rb

```
RSpec.describe User, type: :model do
  # ...
  it { should respond_to(:auth_token) }
  it { should validate_uniqueness_of(:auth_token)}
end
```

We then move to the `user.rb` file and add the necessary code to make our tests pass:

```
/app/models/user.rb
----
class User < ApplicationRecord
  validates :auth_token, uniqueness: true
  # ...
end
----
```

Next we will work on a method that will generate a unique authentication token for each user in order to authenticate them later through the API. So let's build the tests first.

spec/models/user_spec.rb

```
RSpec.describe User, type: :model do
  # ...
  describe "#generate_authentification_token!" do
    it "generates a unique token" do
      @user.generate_authentification_token!
      expect(@user.auth_token).not_to be_nil
    end

    it "generates another token when one already has been taken"
do
      existing_user = FactoryBot.create(:user, auth_token:
"auniquetoken123")
      @user.generate_authentification_token!
      expect(@user.auth_token).not_to eql existing_user
.auth_token
    end
  end
end
```

Before use validate_uniqueness_of method we have to install shoulda-matchers gem. To do so add this gem in Gemfile:

Gemfile

```
# ...
group :test do
  # ...
  gem 'shoulda-matchers'
end
```

And load it in rails_helper.rb:

spec/rails_helper.rb

```
# ...
RSpec.configure do |config|
  # ...
  RSpec.configure do |config|
    config.include(Shoulda::Matchers::ActiveModel, type:
:model)
    config.include(Shoulda::Matchers::ActiveRecord, type:
:model)
  end
end
```

The tests initially fail, as expected:

```
$ bundle exec rspec spec/models/user_spec.rb
.......FF

Failures:

  1) User#generate_authentification_token! generates a unique
token
     Failure/Error: @user.generate_authentification_token!

     NoMethodError:
       undefined method `generate_authentification_token!' for
#<User:0x0000558948d23760>
     # ./spec/models/user_spec.rb:23:in `block (3 levels) in
<top (required)>'

  2) User#generate_authentification_token! generates another
token when one already has been taken
     Failure/Error: @user.generate_authentification_token!

     NoMethodError:
       undefined method `generate_authentification_token!' for
#<User:0x0000558948d18720>
     # ./spec/models/user_spec.rb:29:in `block (3 levels) in
<top (required)>'
```

We are going to hook this `generate_authentication_token!` to a `before_create` callback to warranty every user has an authentication token which does not collides with an existing one. To create the token there are many solutions, I'll go with the `friendly_token` that devise offers already, but I could also do it with the `hex` method from the `SecureRandom` class.

The code to generate the token is fairly simple:

app/models/user.rb

```ruby
class User < ApplicationRecord
  before_create :generate_authentification_token!
  # ...
  def generate_authentification_token!
    begin
      self.auth_token = Devise.friendly_token
    end while self.class.exists?(auth_token: auth_token)
  end
end
```

After that we just need to hook it up to the `before_create` callback:

```
$ bundle exec rspec spec/models/user_spec.rb
.........

Finished in 0.05079 seconds (files took 0.49029 seconds to load)
9 examples, 0 failures
```

As usual, let's commit the changes and move on:

```
$ git add .
$ git commit -m "Adds user authentification token"
```

Sessions controller

Back to the sessions controller the actions we'll be implementing on it are going to be handled as RESTful services: the sign in will be handled by a *POST* request to the create action and the sign out will be handled by a *DELETE* request to the destroy action.

To get started we will start by creating the sessions controller:

```
$ rails generate controller sessions
```

Then we need to move the files into the api/v1 directory, for both on the app and spec folders:

```
$ mv app/controllers/sessions_controller.rb
app/controllers/api/v1
$ mv spec/controllers/sessions_controller_spec.rb
spec/controllers/api/v1
```

After moving the files we have to update them to meet the directory structure we currently have as shown on followed snippets:

app/controllers/api/v1/sessions_controller.rb

```ruby
class Api::V1::SessionsController < ApplicationController
end
```

spec/controllers/api/v1/sessions_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::SessionsController, type: :controller do
end
```

Sign in success

Our first stop will be the create action. But first let's generate our tests:

spec/controllers/api/v1/sessions_controller_spec.rb

```
# ...
RSpec.describe Api::V1::SessionsController, type: :controller do
  describe 'POST #create' do
    before(:each) do
      @user = FactoryBot.create :user
    end

    context 'when the credentials are correct' do
      before(:each) do
        post :create, params: {
          session: { email: @user.email, password: '12345678' }
        }
      end

      it 'returns the user record corresponding to the given
credentials' do
        @user.reload
        expect(json_response[:auth_token]).to eql @user
.auth_token
      end

      it { expect(response.response_code).to eq(200) }
    end

    context 'when the credentials are incorrect' do
      before(:each) do
        post :create, params: {
          session: { email: @user.email, password:
'invalidpassword' }
        }
      end

      it 'returns a json with an error' do
        expect(json_response[:errors]).to eql 'Invalid email or
password'
      end

      it { expect(response.response_code).to eq(422) }
    end
  end
end
```

The tests are pretty straightforward. We simply return the `user` in JSON format if the credentials are correct but if not we just send a JSON with an error message. Next we need to implement the code to make our tests be green. But before that we will add the end points to our `route.rb` file (both the `create` and `destroy` end point).

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # ...
  resources :sessions, :only => [:create, :destroy]
end
```

app/controllers/api/v1/sessions_controller.rb

```
class Api::V1::SessionsController < ApplicationController
  def create
    user_password = params[:session][:password]
    user_email = params[:session][:email]
    user = user_email.present? && User.find_by(email:
user_email)

    if user.valid_password? user_password
      sign_in user
      user.generate_authentification_token!
      user.save
      render json: user, status: 200, location: [:api, user]
    else
      render json: { errors: 'Invalid email or password' },
status: 422
    end
  end
end
```

Before we run our tests it is necessary to add the `devise` test helpers in the `spec_helper.rb` file:

spec/rails_helper.rb

```
# ...
RSpec.configure do |config|
  # ...
  config.include Devise::Test::ControllerHelpers, :type =>
:controller
end
```

Now if we run our tests they should be all passing:

```
$ bundle exec rspec
spec/controllers/api/v1/sessions_controller_spec.rb
....

Finished in 0.06515 seconds (files took 0.49218 seconds to load)
4 examples, 0 failures
```

Now this would be a nice moment to commit the changes:

```
$ git add .
$ git commit -m "Adds sessions controller create action"
```

Sign out

We currently have the `sign in` end point for the API. Now it is time to build a `sign out` url. You might wonder why since we are not handling `sessions` and there is nothing to destroy. In this case we are going to update the authentication token so the last one becomes useless and cannot be used again.

NOTE It is actually not necessary to include this end point, but I do like to include it to expire the authentication tokens.

As usual we start with the tests:

spec/controllers/api/v1/sessions_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::SessionsController, type: :controller do
  # ...
  describe "DELETE #destroy" do

    before(:each) do
      @user = FactoryBot.create :user
      sign_in @user, store: false
      delete :destroy, params: { id: @user.auth_token }
    end

    it { expect(response.response_code).to eq(204) }
  end
end
```

As you can see the `test` is super simple. Now we just need to implement the necessary code to make our tests pass:

app/controllers/api/v1/sessions_controller.rb

```ruby
class Api::V1::SessionsController < ApplicationController
  # ...
  def destroy
    user = User.find_by(auth_token: params[:id])
    user.generate_authentication_token!
    user.save
    head 204
  end
end
```

In this case we are expecting an `id` to be sent on the request which has to correspond to the *user authentication token*. We will add the `current_user` method to handle this smoothly. For now we will just leave it like that.

Take a deep breath, we are almost there! In the meantime commit the changes:

```
$ git add .
$ git commit -m "Adds destroy session action added"
```

Current User

If you have worked with devise before you probably are familiar with the auto-generated methods for handling the authentication filters or getting the user that is currently on session. [4]

In our case we will need to override the `current_user` method to meet our needs, and that is finding the user by the authentication token that is going to be sent on each request to the api. Let me clarify that for you.

Once the client sign ins a user with the correct credentials. The API will return the `authentication token` from that actual user. Each time that client requests for a protected page we will need to fetch the user from that `authentication token` that comes in the request and it could be as a `param` or as a `header`.

In our case we'll be using an `Authorization` header which is commonly used for this type of purpose. I personally find it better because it gives context to the actual request without polluting the URL with extra parameters.

When it comes to authentication I like to add all the related methods into a separate file, and after that just include the file inside the `ApplicationController`. This way it is really easy to test in isolation. Let's create the file under de `controllers/concerns` directory:

```
$ touch app/controllers/concerns/authenticable.rb
```

After that let's create a `concerns` directory under `spec/controllers/` and an `authenticable_spec.rb` file for our authentication tests.

```
$ mkdir spec/controllers/concerns
$ touch spec/controllers/concerns/authenticable_spec.rb
```

As usual we start by writing our tests, in this case for our `current_user` method, which will fetch a user by the authentication token ok the `Authorization` header.

spec/controllers/concerns/authenticable_spec.rb

```ruby
# ...
class Authentication < ActionController::API
  include Authenticable
end

RSpec.describe Authenticable do
  let(:authentication) { Authentication.new }
  subject { authentication }

  describe "#current_user" do
    before do
      @user = FactoryBot.create :user
      request.headers["Authorization"] = @user.auth_token
      authentication.stub(:request).and_return(request)
    end
    it "returns the user from the authorization header" do
      expect(authentication.current_user.auth_token).to eql
@user.auth_token
    end
  end
end
```

NOTE If you are wondering: *"Why the hell we created an Authentication class inside the spec file??"*. The answer is simple: when it comes to test modules I find it easy to include them into a temporary class and stub any other methods I may require later.

Our tests should fail. Let's implement the necessary code:

app/controllers/concerns/authenticable.rb

```ruby
module Authenticable
  # Devise methods overwrites
  def current_user
    @current_user ||= User.find_by(auth_token: request.headers
['Authorization'])
  end
end
```

Now our tests should be green:

```
$ rspec spec/controllers/concerns/authenticable_spec.rb
.

Finished in 0.0149 seconds (files took 0.49496 seconds to load)
1 example, 0 failures
```

Now we just need to include the Authenticable module into the ApplicationController:

app/controllers/application_controller.rb

```ruby
class ApplicationController < ActionController::API
  # ...
  include Authenticable
end
```

This would be a good time to commit the changes:

```bash
$ git add .
$ git commit -m "Adds authenticable module for managing authentication methods"
```

[4] See documentation on this for more details).

Authenticate with token

Authorization is a big part when building applications because in contrary to authentication that allows us to identify the user in the system, authorization help us to define what they can do.

Although we have a good end point for updating the user it has a major security hole: allowing anyone to update any user on the application. In this section we'll be implementing a method that will require the user to be signed in preventing in this way any unauthorized access. We will return a not authorized JSON message along with its corresponding http code.

First we have to add some tests on the `authenticable_spec.rb` for the `authenticate_with_token` method:

spec/controllers/concerns/authenticable_spec.rb

```
# ...
class Authentication < ActionController::API
  include Authenticable
end

RSpec.describe Authenticable do
  # ...
  describe '#authenticate_with_token' do
    before do
      @user = FactoryBot.create :user
      authentication.stub(:current_user).and_return(nil)
      response.stub(:response_code).and_return(401)
      response.stub(:body).and_return({ 'errors' => 'Not
authenticated' }.to_json)
      authentication.stub(:response).and_return(response)
    end

    it 'render a json error message' do
      expect(json_response[:errors]).to eql 'Not authenticated'
    end

    it { expect(response.response_code).to eq(401) }
  end
end
```

As you can see we are using the `Authentication` class again and stubbing the `request` and `response` for handling the expected answer from the server. Now it is time to implement the code to make our tests pass.

app/controllers/concerns/authenticable.rb

```
module Authenticable
  # ...
  def authenticate_with_token!
    unless current_user.present?
      render json: { errors: 'Not authenticated' },
             status: :unauthorized
    end
  end
end
```

At this point we have just built a very simple authorization mechanism to prevent unsigned users from accessing the API. Just update the file `users_controller.rb` with the method `current_user` and prevent access with the command `authenticate_with_token!`!

Let’s commit these changes and keep moving forward:

```
$ git commit -m "Adds the authenticate with token method to
handle access to actions"
```

Authorize actions

It is now time to update our `users_controller.rb` file to deny the access to some of the actions. Also we will implement the `current_user` method on the `update` and `destroy` actions to make sure that the user who is on **session' will be capable only to `update** its data or self `destroy`.

We will start with the `update` action. We will no longer fetch the user by id, instead of that by the `auth_token` on the `Authorization` header provided by the current_user method.

app/controllers/api/v1/users_controller.rb

```ruby
class Api::V1::UsersController < ApplicationController
  # ...
  def update
    # we just change method here
    user = current_user

    if user.update(user_params)
      render json: user, status: 200, location: [:api, user]
    else
      render json: { errors: user.errors }, status: 422
    end
  end
  # ...
end
```

And as you might expect, if we run our users controller specs they should fail:

```
$ rspec spec/controllers/api/v1/users_controller_spec.rb
.......FFFFF.

Failures:

  1) Api::V1::UsersController PUT/PATCH #update when is
successfully updated renders the json representation for the
updated user
     Failure/Error: if user.update(user_params)

     NoMethodError:
       undefined method 'update' for nil:NilClass

    ...
```

The solution is fairly simple: we just need to add the `Authorization` header to the request.

spec/controllers/api/v1/users_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  # ...
  describe 'PUT/PATCH #update' do
    context 'when is successfully updated' do
      before(:each) do
        @user = FactoryBot.create :user
        request.headers['Authorization'] = @user.auth_token
        patch :update, params: { id: @user.id, user: { email:
'newmail@example.com' } }, format: :json
      end
      # ...
    end

    context 'when is not created' do
      before(:each) do
        @user = FactoryBot.create :user
        request.headers['Authorization'] = @user.auth_token
        patch :update, params: { id: @user.id, user: { email:
'bademail.com' } }, format: :json
      end
      # ...
    end
  end
  # ...
end
```

Now the tests should be all green. But wait something does not feel quite right isn't it? We can refactor the line we just added and put it on the `HeadersHelpers` module we build:

spec/support/request_helpers.rb

```ruby
module Request
  # ...
  module HeadersHelpers
    # ...
    def api_authorization_header(token)
      request.headers['Authorization'] = token
    end
  end
end
```

Now each time we need to have the current_user on our specs we simply call the api_authorization_header method. I'll let you do that with the users_controller_spec.rb for the update spec. For the destroy action we will do the same, because we just have to make sure a user is capable to self destroy

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  # ...
  describe 'PUT/PATCH #update' do
    context 'when is successfully updated' do
      before(:each) do
        @user = FactoryBot.create :user
        api_authorization_header @user.auth_token
        patch :update, params: { id: @user.id, user: { email:
'newmail@example.com' } }, format: :json
      end
      # ...
    end

    context 'when is not created' do
      before(:each) do
        @user = FactoryBot.create :user
        api_authorization_header @user.auth_token
        patch :update, params: { id: @user.id, user: { email:
'bademail.com' } }, format: :json
      end
      # ...
    end
  end
  # ...
end
```

Now for the spec file and as mentioned before, we just need to add the api_authorization_header:

app/controllers/api/v1/users_controller.rb

```ruby
class Api::V1::UsersController < ApplicationController
  # ...
  def destroy
    current_user.destroy
    head 204
  end
  # ...
end
```

Now for the spec file and as mentioned before, we just need to add the `api_authorization_header`:

spec/controllers/api/v1/users_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  # ...
  describe 'DELETE #destroy' do
    before(:each) do
      @user = FactoryBot.create :user
      api_authorization_header @user.auth_token
      delete :destroy, params: { id: @user.id }
    end

    it { expect(response.response_code).to eq(204) }
  end
end
```

We should have all of our tests passing. The last step for this section consist on adding the corresponding authorization access for these last two actions.

NOTE	It is common to just prevent the actions on which the user is performing actions on the record itself (in this case the `destroy` and `update` action).

On the `users_controller.rb` we have to filter some these actions to prevent the access

app/controllers/api/v1/users_controller.rb

```
class Api::V1::UsersController < ApplicationController
  before_action :authenticate_with_token!, only: %i[update
destroy]
  respond_to :json
  # ...
end
```

Our tests should still be passing. And from now on every time we want to prevent any action from being trigger we simply add the `authenticate_with_token!` method on a `before_action` hook.

Let’s just commit this:

```
$ git add .
$ git commit -m "Adds authorization for the users controller"
```

Lastly but not least we will finish the chapter by refactoring the `authenticate_with_token!` method. It is really a small enhancement but it will make the method more descriptive. You’ll see what I mean in a minute. But first things first let’s add some specs.

spec/controllers/concerns/authenticable_spec.rb

```ruby
# ...
RSpec.describe Authenticable do
  # ...
  describe '#user_signed_in?' do
    context "when there is a user on 'session'" do
      before do
        @user = FactoryBot.create :user
        authentication.stub(:current_user).and_return(@user)
      end

      it { should be_user_signed_in }
    end

    context "when there is no user on 'session'" do
      before do
        @user = FactoryBot.create :user
        authentication.stub(:current_user).and_return(nil)
      end

      it { should_not be_user_signed_in }
    end
  end
end
```

As you can see we added two simple specs to know whether the user is signed in or not (As I mentioned early it is just for visual clarity). But let's keep going and add the implementation.

app/controllers/concerns/authenticable.rb

```ruby
module Authenticable
  # ...
  def authenticate_with_token!
    unless user_signed_in?
      render json: { errors: 'Not authenticated' },
                status: :unauthorized
    end
  end

  def user_signed_in?
    current_user.present?
  end
end
```

As you can see, now the `authenticate_with_token!` it's easier to read not just for you but for other developers joining the project. This approach has also another side benefit. In any case you want to change or extend how to validate if the user is signed in you can just do it on the `user_signed_in?` method.

Now our tests should be all green:

```
$ rspec spec/controllers/concerns/authenticable_spec.rb
.....

Finished in 0.07415 seconds (files took 0.702 seconds to load)
5 examples, 0 failures
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Adds user_signed_in? method to know whether
the user is logged in or not"
```

Conclusion

Yeah! you made it! you are half way done! Keep up the good work. This chapter was a long and hard one but it is a great step forward on setting a solid mechanism for handling user authentication. We even scratch the surface for simple authorization rules.

In the next chapter we will be focusing on customizing the JSON output for the user with `active_model_serializers` gem and adding a `product` model to the equation by giving the user the ability to create a product and publish it for sale.

User products

On previous chapter we implemented the authentication mechanism who we’ll be using all along the app. Right now we have a very simple implementation of the `user` model but the moment of truth has come where. We will customize the JSON output but also add a second resource: *user products*. These are the items that the user will be selling in the app, and by consequence will be directly associated. If you are familiar with Rails you may already know what I’m talking about. For those who doesn’t known what I’m talking about we will associated the `User` to the `Product` model using the `has_many` and `belongs_to` active record methods.

In this chapter we will build the `Product` model from the ground up associate it with the user and create the necessary end points for any client to access the information.

You can clone the project up to this point:

```
$ git clone --branch chapter6
https://github.com/madeindjs/market_place_api
```

Before we start and as usual when starting new features, we need to branch it out:

```
$ git checkout -b chapter6
```

Product model

We first start by creating Product model then we add some validations to it and finally we will associate it with the `user` model. As the user model the product will be fully tested and will also have an *automatic destruction* if the user in this case is destroyed.

Product bare bones

The product model will need several fields:

- a `price` attribute to hold the product price
- a `published` boolean to know whether the product is ready to sell or not
- a `title` to define a sexy product title
- a `user_id` to associate this particular product to a user

As you may already know we generate it with the `rails generate` command:

```
$ rails generate model Product title:string price:decimal
published:boolean user_id:integer:index
    invoke  active_record
    create    db/migrate/20181218064350_create_products.rb
    create    app/models/product.rb
    invoke    rspec
    create      spec/models/product_spec.rb
    invoke      factory_bot
    create        spec/factories/products.rb
```

As you may notice we also added an `index` option to the `user_id` attribute. This is a good practice when using association keys as it optimizes the query to a database level. It is not compulsory that you do that but I highly recommend it.

The migration file should look like this:

db/migrate/20181218064350_create_products.rb

```
class CreateProducts < ActiveRecord::Migration[5.2]
  def change
    create_table :products do |t|
      t.string :title
      t.decimal :price
      t.boolean :published
      t.integer :user_id

      t.timestamps
    end
    add_index :products, :user_id
  end
end
```

Take note that we set some default values for all of the attributes except the user_id. This way we keep a high consistency level on our database as we don't deal with many NULL values.

Next we will add some basic tests to the Product model. We will just make sure the object responds to the fields we added, as shown next:

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  let(:product) { FactoryBot.build :product }
  subject { product }

  it { should respond_to(:title) }
  it { should respond_to(:price) }
  it { should respond_to(:published) }
  it { should respond_to(:user_id) }
end
```

Remember to migrate the database so we get out tests green:

```
$ rake db:migrate
```

Make sure the tests pass:

```
$ rspec spec/models/product_spec.rb
```

Although our tests are passing we need to do some ground work for the product factory (as for now is all hardcoded). As you recall we have been using Faker to fake the values for our tests models. Now it is time to do the same with the product model.

spec/factories/products.rb

```
FactoryBot.define do
  factory :product do
    title { FFaker::Product.product_name }
    price { rand * 100 }
    published { false }
    user_id { 1 }
  end
end
```

Now each product we create will look a bit more like a real product. We still need to work on the user_id as is hardcoded but we will get to that later.

Product validations

As we saw with the user, validations are an important part when building any kind of application. This will prevent any junk data from being saved onto the database. In the product we have to make sure for example the price is a number and that is not negative.

Also an important thing about validation when working with associations, is in this case to validate that every product has a user, so in this case we need to validate the presence of the user_id. You can see what I'm talking about in next code snippet.

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  # ...
  it { should validate_presence_of :title }
  it { should validate_presence_of :price }
  it { should validate_numericality_of(:price
).is_greater_than_or_equal_to(0) }
  it { should validate_presence_of :user_id }
end
```

Now we need to add the implementation to make the tests pass:

app/models/product.rb

```
class Product < ApplicationRecord
  validates :title, :user_id, presence: true
  validates :price, numericality: { greater_than_or_equal_to: 0
}, presence: true
end
```

Tests are now green:

```
$ rspec spec/models/product_spec.rb
........

Finished in 0.04173 seconds (files took 0.74322 seconds to load)
8 examples, 0 failures
```

We have a bunch of good quality code. Let’s commit it and keep moving:

```
$ git add .
$ git commit -m "Adds product model bare bones along with some
validations"
```

Product/User association

In this section we will be building the association between the product and the user model, we already have the necessary fields, so we just need to update a couple of files and we will be ready to go.

First we need to modify the products factory to relate it to the user. So how do we do that?:

spec/factories/products.rb

```ruby
FactoryBot.define do
  factory :product do
    title { FFaker::Product.product_name }
    price { rand * 100 }
    published { false }
    user
  end
end
```

As you can see we just rename the `user_id` attribute to `user` and we did not specify a value. FactoryBot is smart enough to create a `user` object for every product and associate them automatically. Now we need to add some tests for the association.

spec/models/product_spec.rb

```ruby
# ...
RSpec.describe Product, type: :model do
  # ...
  it { should belong_to :user }
end
```

As you can see the test we added is very simple (thanks to the power of shoulda-matchers). We continue with the implementation now:

app/models/product.rb

```ruby
class Product < ApplicationRecord
  belongs_to :user
  #...
end
```

Remember to run the test we added just to make sure everything is all right:

```
$ rspec spec/models/product_spec.rb
.........

Finished in 0.08815 seconds (files took 0.75134 seconds to load)
9 examples, 0 failures
```

Currently we only have one part of the association, but as you may be wondering already we have to add a `has_many` association to the user model.

First we add the test on the `user_spec.rb` file:

spec/models/user_spec.rb

```
# ...
RSpec.describe User, type: :model do
  # ...
  it { should have_many(:products) }
  # ...
end
```

The implementation on the `user` model is extremely easy:

app/models/user.rb

```
class User < ApplicationRecord
  has_many :products
  # ...
end
```

Now if we run the user specs. They should be all nice and green:

```
$ rspec spec/models/user_spec.rb
..........

Finished in 0.08411 seconds (files took 0.74624 seconds to load)
10 examples, 0 failures
```

Dependency destroy

Something I've seen in other developers code when working with associations, is that they forget about dependency destruction between models. What I mean by this is that if a user is destroyed, the user’s products in this case should be destroyed as well.

So to test this interaction between models, we need a user with a bunch of products, then we destroy that user expecting the products disappear along with it. A simple implementation would look like this:

```
products = user.products
user.destroy
products.each do |product|
  expect(Product.find(product.id)).to raise_error ActiveRecord
::RecordNotFound
end
```

We first save the products into a variable for later access then we destroy the user and loop through the products variable expecting each of the products to raise an exception. Putting everything together should look like the code bellow:

spec/models/user_spec.rb

```
# ...
RSpec.describe User, type: :model do
  # ...
  describe '#products association' do
    before do
      @user.save
      3.times { FactoryBot.create :product, user: @user }
    end

    it 'destroys the associated products on self destruct' do
      products = @user.products
      @user.destroy
      products.each do |product|
        expect { Product.find(product.id) }.to raise_error
ActiveRecord::RecordNotFound
      end
    end
  end
end
```

The necessary code to make tests pass is just an option on the `has_many` association method:

app/models/user.rb

```
class User < ApplicationRecord
  has_many :products, dependent: :destroy
  # ...
end
```

With that code added all of our tests should be passing:

```
$ rspec spec/
...........................................

Finished in 0.44188 seconds (files took 0.8351 seconds to load)
43 examples, 0 failures
```

Let’s commit this and move on to the next sections.

```
$ git add .
$ git commit -m "Finishes modeling the product model along
with user associations"
```

Products endpoints

It is now time to start building the products endpoints. For now we will just build 5 REST actions and some of them will be nested inside the `users` resource. In the next Chapter we will customize the JSON output by implementing the `active_model_serializers` gem.

First we need to create the `products_controller`, and we can easily achieve this with the command below:

```
$ rails generate controller api/v1/products
```

The command above will generate a bunch of files ready to start working, what I mean by this is that it will generate the controller and specs files already scoped to the version 1 of the API.

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
end
```

spec/controllers/api/v1/products_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
end
```

As a warmup we will start nice and easy by building the `show` action for the product.

Show action for products

As usual we begin by adding some product `show` controller specs. The strategy here is very simple: we just need to create a single product and make sure the response from server is what we expect.

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  describe 'GET #show' do
    before(:each) do
      @product = FactoryBot.create :product
      get :show, params: { id: @product.id }
    end

    it 'returns the information about a reporter on a hash' do
      product_response = json_response
      expect(product_response[:title]).to eql @product.title
    end

    it { expect(response.response_code).to eq(200) }
  end
end
```

We then add the code to make the test pass:

app/controllers/api/v1/products_controller.rb

```
class Api::V1::ProductsController < ApplicationController
  def show
    render json: Product.find(params[:id])
  end
end
```

Wait! Don't run the tests yet. Remember we need to add the resource to the `routes.rb` file:

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # ...
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      # ...
      resources :products, only: [:show]
    end
  end
end
```

Now we make sure the tests are nice and green:

```
$ rspec spec/controllers/api/v1/products_controller_spec.rb
..

Finished in 0.05474 seconds (files took 0.75052 seconds to load)
2 examples, 0 failures
```

As you may notice already the specs and implementation are very simple. Actually they behave the same as the users.

Products list

Now it is time to output a list of products, which could be displayed as the market place product catalog. This endpoint is also accessible without credentials, that means we don’t require the user to be logged-in to access the data. As usual we will start writing some specs.

spec/controllers/api/v1/products_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'GET #index' do
    before(:each) do
      4.times { FactoryBot.create :product }
      get :index
    end

    it 'returns 4 records from the database' do
      products_response = json_response
      expect(products_response).to have(4).items
    end

    it { expect(response.response_code).to eq(200) }
  end
end
```

Warning, the `have` we use on previous test was no longer available since Rspec 3.0. We must install one more gem:

Gemfile

```ruby
# ...
group :test do
  # ...
  gem 'rspec-collection_matchers', '~> 1.1'
end
```

Let's move into the implementation, which for now is going to be a sad `all` class method.

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  def index
    render json: Product.all
  end
  #...
end
```

And remember, you have to add the corresponding route:

```
resources :products, only: %i[show index]
```

We are done for now with the public product endpoints. In the sections to come we will focus on building the actions that require a user to be logged in to access them. Said that we are committing this changes and continue.

```
$ git add .
$ git commit -m "Finishes modeling the product model along
with user associations"
```

Creating products

Creating products is a bit tricky because we'll need some extra configuration to give a better structure to this endpoint. The strategy we will follow is to nest the products `create` action into the users which will deliver us a more descriptive endpoint, in this case `/users/:user_id/products`.

So our first stop will be the `products_controller_spec.rb` file.

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'POST #create' do
    context 'when is successfully created' do
      before(:each) do
        user = FactoryBot.create :user
        @product_attributes = FactoryBot.attributes_for :product
        api_authorization_header user.auth_token
        post :create, params: { user_id: user.id, product:
@product_attributes }
      end

      it 'renders the json representation for the product record
just created' do
        product_response = json_response
        expect(product_response[:title]).to eql
@product_attributes[:title]
      end
      it { expect(response.response_code).to eq(201) }
    end

    context 'when is not created' do
      before(:each) do
        user = FactoryBot.create :user
        @invalid_product_attributes = { title: 'Smart TV',
price: 'Twelve dollars' }
        api_authorization_header user.auth_token
        post :create, params: { user_id: user.id, product:
@invalid_product_attributes }
      end

      it 'renders an errors json' do
        product_response = json_response
        expect(product_response).to have_key(:errors)
      end

      it 'renders the json errors on why the user could not be
created' do
        product_response = json_response
        expect(product_response[:errors][:price]).to include 'is
not a number'
      end

      it { expect(response.response_code).to eq(422) }
    end
  end
end
```

Wow! We added a bunch of code but if you recall from previous section the spec actually looks the same as the user create action (but with minor changes). Remember we have this endpoint nested so we need to make sure we send the user_id param on each request as you can see on:

```
post :create, params: { user_id: user.id, product:
@product_attributes }
```

This way we can fetch the user and create the product for that specific user. But wait there is more. If we take this approach we will have to increment the scope of our authorization mechanism because we have to fetch the user from the user_id param. Well in this case and if you remember we built the logic to get the user from the authorization header and assigned it a current_user method. This is rapidly fixable by just adding the authorization header into the request, and fetch that user from it. So let's do that.

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: [:create]
  # ...
  def create
    product = current_user.products.build(product_params)
    if product.save
      render json: product, status: 201, location: [:api,
product]
    else
      render json: { errors: product.errors }, status: 422
    end
  end

  private

  def product_params
    params.require(:product).permit(:title, :price, :published)
  end
end
```

As you can see we are protecting the create action with the authenticate_with_token! method, and on the create action we are building the product in relation to the current_user.

By this point you may be asking yourself *"Well is it really necessary to nest the action? By the end of the day we don't really use the user_id from the URI pattern"*. In my opinion you are totally right. My only argument here is that with this approach the endpoint is way more descriptive from the outside as we are telling the developers that in order to create a product we need a user.

So it is really up to you how you want to organize your resources and expose them to the world, my way is not the only one and it does not mean is the correct one either. In fact I encourage you to play around with different approaches and choose the one that fills your eye.

One last thing before you run your tests, just the necessary route:

config/routes.rb

```ruby
# ...
Rails.application.routes.draw do
  # ...
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      resources :users, only: %i[show create update destroy] do
        resources :products, only: [:create]
      end
      # ...
    end
  end
end
```

Now if you run the tests now, they should be all green:

```
$ rspec spec/controllers/api/v1/products_controller_spec.rb
.........

Finished in 0.21831 seconds (files took 0.75823 seconds to load)
9 examples, 0 failures
```

Updating products

Hopefully by now you understand the logic to build the upcoming actions, in this section we will focus on the `update` action, which will work similarly to the `create` one, we just need to fetch the product from the database and the update it.

We are first add the action to the routes, so we don't forget later:

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # ...
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      resources :users, only: %i[show create update destroy] do
        resources :products, only: %i[create update]
      end
      # ...
    end
  end
end
```

Before we start dropping some tests I just want to clarify that similarly to the `create` action we will scope the product to the `current_user`. In this case we want to make sure the product we are updating is owned by the current user. So we will fetch that product from the `user.products` association provided by Rails.

First we add some specs:

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'PUT/PATCH #update' do
    before(:each) do
      @user = FactoryBot.create :user
      @product = FactoryBot.create :product, user: @user
      api_authorization_header @user.auth_token
    end

    context 'when is successfully updated' do
      before(:each) do
        patch :update, params: { user_id: @user.id, id:
@product.id, product: { title: 'An expensive TV' } }
      end

      it 'renders the json representation for the updated user'
do
```

```
        product_response = json_response
        expect(product_response[:title]).to eql 'An expensive TV'
      end

      it { expect(response.response_code).to eq(200) }
    end

    context 'when is not updated' do
      before(:each) do
        patch :update, params: { user_id: @user.id, id:
@product.id, product: { price: 'two hundred' } }
      end

      it 'renders an errors json' do
        product_response = json_response
        expect(product_response).to have_key(:errors)
      end

      it 'renders the json errors on why the user could not be
created' do
        product_response = json_response
        expect(product_response[:errors][:price]).to include 'is
not a number'
      end

      it { expect(response.response_code).to eq(422) }
    end
  end
end
```

The tests may look complex but take a second peek. They are almost the same we built for users. The only difference here is the nested routes as we saw on previous section, which in this case we need to send the `user_id` as a parameter.

Now let’s implement the code to make our tests pass:

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: %i[create
update]
  # ...
  def update
    product = current_user.products.find(params[:id])
    if product.update(product_params)
      render json: product, status: 200, location: [:api,
product]
    else
      render json: { errors: product.errors }, status: 422
    end
  end
  # ...
end
```

As you can see the implementation is pretty straightforward. We simply fetch the product from the `current_user` and simply update it. We also added this action to the `before_action` hook to prevent any unauthorized user to update a product.

Now if we run the tests, they should be all green:

```
$ rspec spec/controllers/api/v1/products_controller_spec.rb
..............

Finished in 0.24404 seconds (files took 0.75973 seconds to load)
14 examples, 0 failures
```

Destroying products

Our last stop for the products endpoints will be the `destroy` action. You might now imagine how this would look like. The strategy in here will be pretty similar to the create and update action (which means we are going to nest the route into the `users` resources) then fetch the product from the `user.products` association and finally destroy it, returning a `204` code.

Let's start again by adding the route name to the routes file:

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # ...
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      resources :users, only: %i[show create update destroy] do
        resources :products, only: %i[create update destroy]
      end
      # ...
    end
  end
end
```

After this, we have to add some tests as shown on this code snippet:

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'DELETE #destroy' do
    before(:each) do
      @user = FactoryBot.create :user
      @product = FactoryBot.create :product, user: @user
      api_authorization_header @user.auth_token
      delete :destroy, params: { user_id: @user.id, id:
@product.id }
    end

    it { expect(response.response_code).to eq(204) }
  end
end
```

Now we simply add the necessary code to make the tests pass:

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: %i[create
update destroy]
  # ...
  def destroy
    product = current_user.products.find(params[:id])
    product.destroy
    head 204
  end
  # ...
end
```

As you can see the three-line implementation does the job. We can run the tests to make sure everything is good and after that we will commit the changes as we added a bunch of new code. Also make sure you hook this action to the `before_action` callback as with the `update` action.

```
$ rspec spec/controllers/api/v1/products_controller_spec.rb
...............

Finished in 0.25959 seconds (files took 0.80248 seconds to load)
15 examples, 0 failures
```

Let’s commit the changes:

```
$ git add .
$ git commit -m "Adds the products create, update and destroy
action nested on the user resources"
```

Feed the database

Before we continue with more code let’s populate the database with some fake data. Thankfully we have some factories that should do the work for us. So let’s do use them.

First we run the rails console command from the Terminal:

```
$ rails console
```

We then create a bunch of product objects with the FactoryBot gem:

```
Loading development environment (Rails 5.2.1)
2.5.3 :001 > 20.times { FactoryBot.create :product }
```

Oops, you probably have some errors showing up:

```
Traceback (most recent call last):
        3: from (irb):1
        2: from (irb):1:in `times'
        1: from (irb):1:in `block in irb_binding'
NameError (uninitialized constant FactoryBot)
```

This is because we are running the console on development environment but that does not make sense with our Gemfile, which currently looks like this:

Gemfile

```
# ...
group :test do
  gem 'factory_bot_rails'
  gem 'ffaker', '~> 2.10'
  gem 'rspec-collection_matchers', '~> 1.1'
  gem 'rspec-rails', '~> 3.8'
  gem 'shoulda-matchers'
end
```

You see where the problem is? If you pay attention you will notice that the factory_bot_rails gem is only available for the test environment but no for the development one (which is what we need). This can be fix really fast:

Gemfile

```ruby
# ...
group :development, :test do
  gem 'factory_bot_rails'
  gem 'ffaker', '~> 2.10'
end

group :test do
  # ...
end
```

Notice the we moved the `ffaker` gem to the shared group as we use it inside the factories we describe earlier. Now just run the `bundle` command to update the libraries. Then build the products you want like so:

```
$ rails console
Loading development environment (Rails 5.2.1)
2.5.3 :001 > 20.times { FactoryBot.create :product }
```

From now on you will be able to create any object from factories such as users, products, orders, etc. So let's commit this tiny change:

```bash
$ git add .
$ git commit -m "Updates test environment factory gems to work
on development"
```

Conclusion

On the next chapter we will focus on customizing the output from the `user` and `product` models using the active model serializers gem. It will help us to easily filter attributes to display (or handle associations as embedded objects for example).

I hope you have enjoyed this chapter. It is a long one but the code we put together is an excellent base for the core app.

JSON with Active Model Serializers

In previous chapter we added a products resource to the application and built all the necessary endpoints up to this point. We also associated the product model with the user and protected some of the products_controller actions on the way. By now you should feel really happy with all the work but we still have to do some heavy lifting. Currently we have something which look like this:

```
{
  "products": [
      {
          "id": 1,
          "title": "Tag Case",
          "price": "98.7761933800815",
          "published": false,
          "user_id": 1,
          "created_at": "2018-12-20T12:47:26.686Z",
          "updated_at": "2018-12-20T12:47:26.686Z"
      },
    ]
}
```

It doesn’t look nice, as the JSON output should render just an array of products, with the products as the root key. Something like this:

```
{
  "products": [
      {
          "id": 1,
          "title": "Tag Case",
      },
    ]
}
```

This is further explained in the JSON API website.

> A collection of any number of resources SHOULD be represented as an array of resource objects or IDs, or as a single "collection object"

I highly recommend you go and bookmark this reference. It is amazing and will cover some points I might not.

In this chapter we will customize the JSON output using the active_model_serializers gem, for more information you can review the repository on GitHub. I'll cover some points in here from installation to implementation but I do recommend you check the gem docs on your free time.

You can clone the project up to this point with:

```
$ git clone https://github.com/madeindjs/market_place_api.git
-b chapter6
```

Let's branch out this chapter with:

```
$ git checkout -b chapter7
```

Setting up the gem

If you have been following the tutorial all along. you should already have the gem installed. But in case you just landed here I'll walk you through the setup.

Add the following line to your `Gemfile`:

Gemfile

```
# ...
gem 'active_model_serializers', '~> 0.10.8'
```

Run the `bundle install` command to install the gem and that is it. You should be all set to continue with the tutorial.

Serialise the user model

First we need to add a user_serializer file. We can do it manually but the gem already provides a command line interface to do so:

```
$ rails generate serializer user
  create  app/serializers/user_serializer.rb
```

This created a file called user_serializer under the app/serializers directory, which should look like this:

app/serializers/user_serializer.rb

```
class UserSerializer < ActiveModel::Serializer
  attributes :id
end
```

By now we should have some failing tests. Go ahead and try it:

```
$ rspec spec/controllers/api/v1/users_controller_spec.rb
F.F....F.....
```

NOTE In this case, Rails will look for a serializer named PostSerializer, and if it exists, use it to serialize the Post. This also works with respond_with, which uses to_json under the hood. Also note that any options passed to render :json will be passed to your serializer and available as @options inside. This means that no matter if we are using the render json method or respond_with, from now on Rails will look for the corresponding serializer first.

Now back to the specs. You can see that for some reason the response it's not quite what we are expecting, and that is because the gem encapsulates the model into a javascript object with the model name as the root (in this case user).

So in order to make the tests pass we just need to add the attributes to serialize into the user_serializer.rb and update the users_controller_spec.rb file:

app/serializers/user_serializer.rb

```
class UserSerializer < ActiveModel::Serializer
  attributes :id, :email, :created_at, :updated_at, :auth_token
end
```

Now if you run the tests now, they should be all green:

```
$ rspec spec/controllers/api/v1/users_controller_spec.rb
.............

Finished in 0.16712 seconds (files took 0.80637 seconds to load)
13 examples, 0 failures
```

Let's commit the changes:

```
$ git add .
$ git commit -am "Adds user serializer for customizing the json
output"
```

We can also test the serializer objects as shown on the documentation but I'll let that to you to decide whether or not to test.

Serialize the product model

Now that we kind of understand how the serializers gem works it is time to customize the products output. The first step and as with the user we need a product serializer. So let’s do that:

```
$ rails generate serializer product
    create  app/serializers/product_serializer.rb
```

Now let’s add the attributes to serialize for the product just as we did it with the user back in previous section:

app/serializers/product_serializer.rb

```
class ProductSerializer < ActiveModel::Serializer
  attributes :id, :title, :price, :published
end
```

And that’s it. This is no more complicated as this. You can run all test suite but it will be green. Let’s commit the changes and move on onto next section.

```
$ git add .
$ git commit -a "Adds product serializer for custom json
output"
```

Serializing associations

We have been working with serializers and you may notice that it is quite simple. In some cases the hard decision is how to name your endpoints, or how to structure the JSON output, so your solution is kept through time.

When working with and API and associations between models there are many approaches you can take. Here I will explain what I found works for me and I let you judge. In this section we will extend our API to handle the product-user association. I'll also explain some of the common mistakes or holes in which you can fall into.

Just to recap, we have a `has_many` type association between the user and product model. Check theses code snippets.

app/models/user.rb

```ruby
class User < ApplicationRecord
  has_many :products, dependent: :destroy
  # ...
end
```

app/models/product.rb

```ruby
class Product < ApplicationRecord
  belongs_to :user
  # ...
end
```

This is important because sometimes to save some requests from being placed it is a good idea to embed objects into other objects. This will make the output a bit heavier but when fetching many records this can save you from a huge bottleneck. Let me explain with a use case for the actual application as shown next.

Use case of nested objects associations

Imagine a scenario where you are fetching the products from the API. In this scenario you need to display some of the user info.

One possible solution to this would be to add the `user_id` attribute to the `product_serializer` so we can fetch the corresponding user later. This might sound like a good idea but if you care about performance (or your database transactions are not fast enough) you should reconsider this approach. You have to realize that for every product you fetch you'll have to request its corresponding user.

When facing this problem I've come with two possible alternatives:

- One good solution (in my opinion) is to embed the user ids related to the products into a meta attribute. So we have a JSON output like:

```
{
  "meta": { "user_ids": [1,2,3] },
  "products": [
  ]
}
```

This might need some further configuration on the user's endpoint, so the client can fetch those users from those `user_ids`.

- Another solution (the one which I'll be using here) is to embed the user object into de product object. This can make the first request a bit slower but this way the client does not need to make another extra request. An example of the expected output is presented below:

```
{
  "products":
  [
      {
          "id": 1,
          "title": "Digital Portable System",
          "price": "25.0277354166289",
          "published": false,
          "user": {
             "id": 2,
             "email": "stephany@lind.co.uk",
             "created_at": "2014-07-29T03:52:07.432Z",
             "updated_at": "2014-07-29T03:52:07.432Z",
             "auth_token": "Xbnzbf3YkquUrF_1bNkZ"
           }
      }
    ]
}
```

So we’ll be embedding the user object into the product. Let’s start by adding some tests. We will just modify the `show` and `index` endpoints spec.

spec/controllers/api/v1/products_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'has the user as a embedded object' do
      expect(json_response[:user][:email]).to eql @product.user
.email
    end
  end

  describe 'GET #index' do
    # ...
    it 'returns the user object into each product' do
      json_response.each do |product_response|
        expect(product_response[:user]).to be_present
      end
    end
  end
  # ...
end
```

The implementation is really easy. We just need to add one line to the product serializer:

app/serializers/product_serializer.rb

```ruby
class ProductSerializer < ActiveModel::Serializer
  attributes :id, :title, :price, :published
  has_one :user
end
```

Now if we run our tests, they should be all green:

```
$ rspec spec
............................................................

Finished in 0.57068 seconds (files took 0.67788 seconds to load)
60 examples, 0 failures
```

Embedding products on users

By now you may be asking yourself if you should embed the products into the user the same as the the section above. Although it may sound fair, this can take to severe optimization problems, as you could be loading huge amounts of information and it is really easy to fall into the Circular Reference problem [5].

But don’t worry not all is lost. We can easily solve this problem, and this is by embedding just the ids from the products into the user, giving your API a better performance and avoid loading extra data. So in this section we will extend our products index endpoint to deal with a product_ids parameter and format the JSON output accordingly.

First we make sure the product_ids it is part of the user serialized object:

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'has the product ids as an embedded object' do
      expect(json_response[:product_ids]).to eql []
    end
  end
  # ...
end
```

The implementation is very simple, as described by the active_model_serializers gem documentation:

app/serializers/user_serializer.rb

```
class UserSerializer < ActiveModel::Serializer
  attribute :product_ids do
    object.products.map(&:id)
  end
  # ...
end
```

We should have our tests passing:

```
$ rspec spec/controllers/api/v1/users_controller_spec.rb
..............

Finished in 0.16791 seconds (files took 0.65902 seconds to load)
14 examples, 0 failures
```

Now we need to extend the `index` action from the `products_controller` so it can handle the product_ids parameter and display the scoped records. Let’s start by adding some specs:

spec/controllers/api/v1/products_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'GET #index' do
    before(:each) do
      4.times { FactoryBot.create :product }
      get :index
    end

    context 'when is not receiving any product_ids parameter' do
      before(:each) do
        get :index
      end

      it 'returns 4 records from the database' do
        expect(json_response).to have(4).items
      end

      it 'returns the user object into each product' do
        json_response.each do |product_response|
          expect(product_response[:user]).to be_present
        end
      end

      it { expect(response.response_code).to eq(200) }
    end

    context 'when product_ids parameter is sent' do
      before(:each) do
        @user = FactoryBot.create :user
        3.times { FactoryBot.create :product, user: @user }
        get :index, params: { product_ids: @user.product_ids }
      end

      it 'returns just the products that belong to the user' do
        json_response.each do |product_response|
          expect(product_response[:user][:email]).to eql @user
.email
        end
      end
    end
  end
  # ...
end
```

As you can see from previous code we just wrapped the index action into two separate contexts: one which will receive the product_ids, and the old one we had which does not. Let's add the necessary code to make the tests pass:

app/controllers/api/v1/products_controller.rb

```
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: %i[create
update destroy]

  def index
    products = params[:product_ids].present? ? Product.find
(params[:product_ids]) : Product.all
    render json: products
  end
  # ...
end
```

As you can see the implementation is super simple. We simply just fetch the products from the product_ids params in case they are present, otherwise we just fetch all of them. Let's make sure the tests are passing:

```
$ rspec spec/controllers/api/v1/products_controller_spec.rb
..................

Finished in 0.35027 seconds (files took 0.65369 seconds to load)
18 examples, 0 failures
```

Let's commit the changes:

```
$ git commit -am "Embeds the products_ids into the user
serialiser and fetches the correct products from the index
action endpoint"
```

[5] in short loops the program until it runs out of memory and throws you and error or never respond you at all

Searching products

In this last section we will keep up the heavy lifting on the `index` action for the products controller by implementing a super simple search mechanism to let any client filter the results. This section is optional as it's not going to have impact on any of the modules in the app. If you want to practice more with TDD and keep the brain warm I recommend you complete this last step.

I've been using Ransack to build advance search forms extremely fast, but as this is an education tool (or at least I consider it), and the search we'll be performing is really simple, I think we can build a simple search engine, we just need to consider the criteria by which we are going to filter the attributes. Hold tight to your seats this is going to be a rough ride.

We will filter the products by the following criteria:

- By a title pattern
- By price
- Sort by creation

This may sound short and easy but believe me it will give you a headache if you don't plan it.

By keyword

We will create a scope to find the records which match a particular pattern of characters, let's called it `filter_by_title`, let's add some specs first:

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  # ...
  describe '.filter_by_title' do
    before(:each) do
      @product1 = FactoryBot.create :product, title: 'A plasma
TV'
      @product2 = FactoryBot.create :product, title: 'Fastest
Laptop'
      @product3 = FactoryBot.create :product, title: 'CD player'
      @product4 = FactoryBot.create :product, title: 'LCD TV'
    end

    context "when a 'TV' title pattern is sent" do
      it 'returns the 2 products matching' do
        expect(Product.filter_by_title('TV')).to have(2).items
      end

      it 'returns the products matching' do
        expect(Product.filter_by_title('TV').sort).to
match_array([@product1, @product4])
      end
    end
  end
end
```

The caveat in here is to make sure no matter the case of the title sent we have to sanitize it to any case in order to make the appropriate comparison in this case we'll use the lower case approach. Let's implement the necessary code:

app/models/product.rb

```
class Product < ApplicationRecord
  # ...
  scope :filter_by_title, lambda { |keyword|
    where('lower(title) LIKE ?', "%#{keyword.downcase}%")
  }
end
```

The implementation above should be enough to make the tests pass:

```
$ rspec spec/models/product_spec.rb
...........

Finished in 0.17178 seconds (files took 3.59 seconds to load)
11 examples, 0 failures
```

By price

In order to filter by price things can get a little bit tricky but actually it is very easy. We will break the logic to filter by price into two different methods: one which will fetch the products greater than the price received and the other one to look for the ones under that price. By doing this we keep everything really flexible and we can easily test the scopes.

Let’s start by building the above_or_equal_to_price scope specs:

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  # ...
  describe '.above_or_equal_to_price' do
    before(:each) do
      @product1 = FactoryBot.create :product, price: 100
      @product2 = FactoryBot.create :product, price: 50
      @product3 = FactoryBot.create :product, price: 150
      @product4 = FactoryBot.create :product, price: 99
    end

    it 'returns the products which are above or equal to the
price' do
      expect(Product.above_or_equal_to_price(100).sort).to
match_array([@product1, @product3])
    end
  end
end
```

The implementation is extremely simple:

app/models/product.rb

```
class Product < ApplicationRecord
  # ...
  scope :above_or_equal_to_price, lambda { |price|
    where('price >= ?', price)
  }
end
```

That should be sufficient. Let’s just verify everything is ok:

```
$ rspec spec/models/product_spec.rb
............

Finished in 0.1566 seconds (files took 0.64782 seconds to load)
12 examples, 0 failures
```

You can now imagine how the opposite method will behave. Let’s add the specs:

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  # ...
  describe '.below_or_equal_to_price' do
    before(:each) do
      @product1 = FactoryBot.create :product, price: 100
      @product2 = FactoryBot.create :product, price: 50
      @product3 = FactoryBot.create :product, price: 150
      @product4 = FactoryBot.create :product, price: 99
    end

    it 'returns the products which are above or equal to the
price' do
      expect(Product.below_or_equal_to_price(99).sort).to
match_array([@product2, @product4])
    end
  end
end
```

And now the implementation:

app/models/product.rb

```ruby
class Product < ApplicationRecord
  # ...
  scope :below_or_equal_to_price, lambda { |price|
    where('price <= ?', price)
  }
end
```

For our sake let's run the tests and verify everything is nice and green:

```
$ rspec spec/models/product_spec.rb
.............

Finished in 0.18008 seconds (files took 0.6544 seconds to load)
13 examples, 0 failures
```

As you can see we have not gotten in a lot of trouble. Let's just add another scope to sort the records by date of last update. This is because in case the proprietary of the product decides to update some of the data, the client always fetches the most updated records.

Sort by creation

This scope is super easy, let's add some specs first:

spec/models/product_spec.rb

```ruby
# ...
RSpec.describe Product, type: :model do
  # ...
  describe '.recent' do
    before(:each) do
      @product1 = FactoryBot.create :product, price: 100
      @product2 = FactoryBot.create :product, price: 50
      @product3 = FactoryBot.create :product, price: 150
      @product4 = FactoryBot.create :product, price: 99

      # we will touch some products to update them
      @product2.touch
      @product3.touch
    end

    it 'returns the most updated records' do
      expect(Product.recent).to match_array([@product3,
@product2, @product4, @product1])
    end
  end
end
```

And now the code:

app/models/product.rb

```ruby
class Product < ApplicationRecord
  # ...
  scope :recent, lambda {
    order(:updated_at)
  }
end
```

All of our tests should be green:

```
$ rspec spec/models/product_spec.rb
.............

Finished in 0.18008 seconds (files took 0.6544 seconds to load)
13 examples, 0 failures
```

Now it would be a good time to commit the changes as we are done adding scopes:

```
$ git commit -am "Adds search scopes on the product model"
```

Search engine

Now that we have the ground base for the search engine we'll be using in the app it is time to implement a simple but powerful search method, which will handle all the logic for fetching product records.

The method will consist on chaining all of the scopes we previously built and return the expected search. Let's start by adding some tests:

spec/models/product_spec.rb

```
# ...
RSpec.describe Product, type: :model do
  # ...
  describe '.search' do
    before(:each) do
      @product1 = FactoryBot.create :product, price: 100, title:
'Plasma tv'
      @product2 = FactoryBot.create :product, price: 50, title:
'Videogame console'
      @product3 = FactoryBot.create :product, price: 150, title:
'MP3'
      @product4 = FactoryBot.create :product, price: 99, title:
'Laptop'
    end

    context "when title 'videogame' and '100' a min price are
set" do
      it 'returns an empty array' do
        search_hash = { keyword: 'videogame', min_price: 100 }
        expect(Product.search(search_hash)).to be_empty
      end
    end

    context "when title 'tv', '150' as max price, and '50' as
min price are set" do
      it 'returns the product1' do
        search_hash = { keyword: 'tv', min_price: 50,
max_price: 150 }
```

```
        expect(Product.search(search_hash)).to match_array
([@product1])
      end
    end

    context 'when an empty hash is sent' do
      it 'returns all the products' do
        expect(Product.search({})).to match_array([@product1,
@product2, @product3, @product4])
      end
    end

    context 'when product_ids is present' do
      it 'returns the product from the ids' do
        search_hash = { product_ids: [@product1.id, @product2.
id] }
        expect(Product.search(search_hash)).to match_array
([@product1, @product2])
      end
    end
  end
end
```

We added a bunch of code but the implementation is very easy (you'll see). You can go further and add some more specs. In my case I did not find it necessary.

app/models/product.rb

```ruby
class Product < ApplicationRecord
  # ...
  def self.search(params = {})
    products = params[:product_ids].present? ? Product.find
(params[:product_ids]) : Product.all

    products = products.filter_by_title(params[:keyword]) if
params[:keyword]
    products = products.above_or_equal_to_price(params
[:min_price].to_f) if params[:min_price]
    products = products.below_or_equal_to_price(params
[:max_price].to_f) if params[:max_price]
    products = products.recent(params[:recent]) if params
[:recent].present?

    products
  end
end
```

It is important to notice that we return the products as an `ActiveRelation` object so we can further chain more methods in case we need so (or paginate them which we will see on the last chapters). We just need to update the products controller index action to fetch the products from the `search` method:

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: %i[create
update destroy]

  def index
    render json: Product.search(params)
  end
  # ...
end
```

We can run the whole test suite to make sure the app is healthy up to this point:

```
$ rspec spec
.................................................................
......

Finished in 1.49 seconds (files took 6.53 seconds to load)
71 examples, 0 failures
```

Let’s commit theses changes:

```
$ git commit -am "Adds search class method to filter products"
```

Conclusion

On chapters to come, we will start building the `Order` model, associate it with users and products, which so far and thanks to the `active_model_serializers` gem, it’s been easy.

This was a long chapter, you can sit back, rest and look how far we got. I hope you are enjoying what you got until now, it will get better. We still have a lot of topics to cover one of them is optimization and caching.

Placing Orders

Back in previous chapter we handle associations between the product and user models, and how to serialize them in order to scale fast and easy. Now it is time to start placing orders which is going to be a more complex situation. We will handle associations between three models and we have to be smart enough to handle the JSON output we are delivering.

In this chapter we will make several things which I list below:

1. Create an `Order` model with its corresponding specs
2. Handle JSON output association between the order user and product models
3. Send a confirmation email with the order summary

So now that we have everything clear, we can get our hands dirty. You can clone the project up to this point with:

```
$ git clone --branch chapter7
https://github.com/madeindjs/market_place_api
```

Let's create a branch to start working:

```
$ git checkout -b chapter8
```

Modeling order

If you remember associations model, the Order model is associated with users and products at the same time. It is actually really simply to achieve this in Rails. The tricky part is whens comes to serializing this objects. I talk about more about this in a next section.

Let’s start by creating the order model, with a special form:

```
$ rails generate model order user:references total:decimal
```

The command above will generate the order model, but I’m taking advantage of the references method to create the corresponding foreign key for the order to belong to a user, it also adds the belongs_to directive into the order model. Let’s migrate the database and jump into the order_spec.rb file.

```
$ rake db:migrate
```

Now it is time to drop some tests into the order_spec.rb file:

spec/models/order_spec.rb

```
# ...
RSpec.describe Order, type: :model do
  let(:order) { FactoryBot.build :order }
  subject { order }

  it { should respond_to(:total) }
  it { should respond_to(:user_id) }
  it { should validate_presence_of :user_id }
  it { should validate_presence_of :total }
  it { should validate_numericality_of(:total
).is_greater_than_or_equal_to(0) }
  it { should belong_to :user }
end
```

The implementation is fairly simple:

app/models/order.rb

```
class Order < ApplicationRecord
  belongs_to :user
  validates :total, numericality: { greater_than_or_equal_to: 0
}
  validates :total, presence: true
  validates :user_id, presence: true
end
```

If we run the tests now they should be all green:

```
$ rspec spec/models/order_spec.rb
......

Finished in 0.16229 seconds (files took 4.08 seconds to load)
6 examples, 0 failures
```

Orders and Products

We need to setup the association between the `order` and the `product` and this is build with a `has-many-to-many` association. As many products will be placed on many orders and the orders will have multiple products. So in this case we need a model in the middle which will join these two other objects and map the appropriate association.

Let’s generate this model:

```
$ rails generate model placement order:references
product:references
```

Let’s migrate the database:

```
$ rake db:migrate
```

Let’s add the order association specs first:

spec/models/order_spec.rb

```ruby
# ...
RSpec.describe Order, type: :model do
  # ...
  it { should have_many(:placements) }
  it { should have_many(:products).through(:placements) }
end
```

The implementation is like so:

app/models/order.rb

```ruby
class Order < ApplicationRecord
  has_many :placements
  has_many :products, through: :placements
  # ...
end
```

Now it is time to jump into the `product-placement` association:

spec/models/product_spec.rb

```ruby
# ...
RSpec.describe Product, type: :model do
  # ...
  it { should have_many(:placements) }
  it { should have_many(:orders).through(:placements) }
  # ...
end
```

Let's add the code to make it pass:

app/models/product.rb

```ruby
class Product < ApplicationRecord
  belongs_to :user
  has_many :placements
  has_many :orders, through: :placements
  # ...
end
```

And lastly but not least, the `placement` specs:

spec/models/placement_spec.rb

```ruby
# ...
RSpec.describe Placement, type: :model do
  let(:placement) { FactoryBot.build :placement }
  subject { placement }
  it { should respond_to :order_id }
  it { should respond_to :product_id }
  it { should belong_to :order }
  it { should belong_to :product }
end
```

If you have been following the tutorial so far the implementation is already there because of the `references` type we pass on the model command generator. We should add the inverse option to the `placement` model for each `belongs_to` call. This gives a little boost when referencing the parent object.

app/models/placement.rb

```ruby
class Placement < ApplicationRecord
  belongs_to :order, inverse_of: :placements
  belongs_to :product, inverse_of: :placements
end
```

Let's run the `models` spec and make sure everything is green:

```
$ rspec spec/models
...........................................

Finished in 0.53127 seconds (files took 0.73125 seconds to load)
43 examples, 0 failures
```

Now that everything is nice and green let's commit the changes and continue.

```
$ git add .
$ git commit -m "Associates products and orders with a
placements model"
```

User orders

We are just missing one little but very important part, which is to relate the user to the orders. But we did no complete the implementation. So let's do that. First open the `user_model_spec.rb` file to add the corresponding tests:

spec/models/user_spec.rb

```ruby
# ...
RSpec.describe User, type: :model do
  # ...
  it { should have_many(:orders) }
  # ...
end
```

And then just add the implementation, which is super simple:

app/models/user.rb

```ruby
class User < ApplicationRecord
  # ...
  has_many :orders, dependent: :destroy
  # ...
end
```

You can run the tests for both files, and they should be all nice and green:

```
$ rspec spec/models/{order,user}_spec.rb
....................

Finished in 0.14279 seconds (files took 0.72848 seconds to load)
20 examples, 0 failures
```

Let's commit this small changes and move next:

```
$ git add .
$ git commit -m 'Adds user order has many relation'
```

Exposing the order model

It is now time to prepare the orders controller to expose the correct order object, and if you recall past chapters, with ActiveModelSerializers this is really easy.

> But wait, what are we suppose to expose?

You may be wondering. And you are right. Let's first define which actions are we going to build up:

1. An index action to retrieve the current user orders
2. A show action to retrieve a particular order from the current user
3. A create action to actually place the order

Let's start with the `index` action, so first we have to create the orders controller.

```
$ rails g controller api/v1/orders
```

Up to this point and before start typing some code we have to ask ourselves:

> Should I leave my order endpoints nested into the `UsersController`, or should I isolate them?

The answer is really simple. I would say it depends on how much information in this case in particular you want to expose to the developer, not from a JSON output point of view, but from the URI format.

I'll nest the routes, because I like to give this type of information to the developers, as I think it gives more context to the request itself. Let's start by dropping some tests:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  describe 'GET #index' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token
      4.times { FactoryBot.create :order, user: current_user }
      get :index, params: { user_id: current_user.id }
    end

    it 'returns 4 order records from the user' do
      expect(json_response).to have(4).items
    end

    it { expect(response.response_code).to eq(200) }
  end
end
```

If we run the test suite now, as you may expect, both tests will fail, because have not even set the correct routes, nor the action. So let's start by adding the routes:

config/routes.rb

```ruby
# ...
Rails.application.routes.draw do
  # ...
  namespace :api, defaults: { format: :json }, constraints: {
subdomain: 'api' } do
    scope module: :v1, constraints: ApiConstraints.new(version:
1, default: true) do
      resources :users, only: %i[show create update destroy] do
        # ...
        resources :orders, only: [:index]
      end
      # ...
    end
  end
end
```

Now it is time for the orders controller implementation:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  before_action :authenticate_with_token!

  def index
    render json: current_user.orders
  end
end
```

And now all of our tests should pass:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
..

Finished in 0.07943 seconds (files took 0.7232 seconds to load)
2 examples, 0 failures
```

We like our commits very atomic, so let's commit this changes:

```
$ git add .
$ git commit -m "Adds the show action for order"
```

Render a single order

As you may imagine already, this endpoint is super easy, we just have to set up some configuration(routes, controller action) and that would be it for this section.

Let's start by adding some specs:

spec/controllers/api/v1/orders_controller_spec.rb

```
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...
  describe 'GET #show' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token
      @order = FactoryBot.create :order, user: current_user
      get :show, params: { user_id: current_user.id, id:
@order.id }
    end

    it 'returns the user order record matching the id' do
      expect(json_response[:id]).to eql @order.id
    end

    it { expect(response.response_code).to eq(200) }
  end
end
```

Let's add the implementation to make our tests pass. On the `routes.rb` file add the show action to the orders resources:

config/routes.rb

```
# ...
Rails.application.routes.draw do
  # ...
  resources :orders, only: [:index, :show]
  # ...
end
```

And the the implementation should look like this:

app/controllers/api/v1/orders_controller.rb

```
class Api::V1::OrdersController < ApplicationController
  # ...
  def show
    render json: current_user.orders.find(params[:id])
  end
end
```

Our tests should be all green:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
....

Finished in 0.12767 seconds (files took 0.73322 seconds to load)
4 examples, 0 failures
```

Let's commit the changes and move onto the create order action:

```
$ git add .
$ git commit -m "Adds the show action for order"
```

Placing and order

It is now time to let the user place some orders, this will add some complexity to the whole application. But don't worry we will go one step at a time to keep things simple.

Before start this feature, let's sit back and think about the implications of creating an order in the app. I'm not talking about implementing a transactions service like Stripe or Braintree, but things like handling out of stock products, decrementing the product inventory, add some validation for the order placement to make sure there is enough products by the time the order is place. Did you already detected that?, it may look like we are way down on the hill, but believe, you are closer than you think, and is not as hard as it sounds.

For now let's keep things simple an assume we always have enough products to place any number of orders, we just care about the server response for now.

If you recall the `order` model on Section 8.1 we need basically 3 things, a total for the order, the user who is placing the order and the products for the order. Given that information we can start adding some specs:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...
  describe 'POST #create' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryBot.create :product
      product_2 = FactoryBot.create :product
      order_params = { total: 50, user_id: current_user.id,
product_ids: [product_1.id, product_2.id] }
      post :create, params: { user_id: current_user.id, order:
order_params }
    end

    it 'returns the just user order record' do
      expect(json_response[:id]).to be_present
    end

    it { expect(response.response_code).to eq(201) }
  end
end
```

As you can see we are creating a `order_params` variable with the order data. Can you see the problem here? If not, I'll explain it later. Let's just add the necessary code to make this test pass.

First we need to add the action to the resources on the routes file:

config/routes.rb

```ruby
# ...
Rails.application.routes.draw do
  # ...
  resources :orders, only: %i[index show create]
  # ...
end
```

Then the implementation which is easy:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  # ...
  def create
    order = current_user.orders.build(order_params)

    if order.save
      render json: order, status: 201, location: [:api,
current_user, order]
    else
      render json: { errors: order.errors }, status: 422
    end
  end

  private

  def order_params
    params.require(:order).permit(:total, :user_id, product_ids:
[])
  end
end
```

And now our tests should all be green:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
......

Finished in 0.16817 seconds (files took 0.64624 seconds to load)
6 examples, 0 failures
```

Ok, so we have everything nice and green. We now should move on to the next chapter right? Let me stop you right there. We have some serious errors on the app, and they are not related to the code itself but on the business part.

Not because the tests are green, it means the app is filling the business part of the app. I wanted to bring this up because in many cases is super easy to just receive params and build objects from those params thinking that we are always receiving the correct data. In this particular case we cannot rely on that, and the easiest way to see this, is that we are letting the client to set the order total, yeah crazy!

We have to add some validations or a callback to calculate the order total an set it through the model. This way we don't longer receive that total attribute and have complete control on this attribute. So let's do that.

We first need to add some specs for the order model:

spec/models/order_spec.rb

```
# ...
RSpec.describe Order, type: :model do
  # ...
  describe '#set_total!' do
    before(:each) do
      product_1 = FactoryBot.create :product, price: 100
      product_2 = FactoryBot.create :product, price: 85

      @order = FactoryBot.build :order, product_ids: [product_1
.id, product_2.id]
    end

    it 'returns the total amount to pay for the products' do
      expect { @order.set_total! }.to change { @order.total
}.from(0).to(185)
    end
  end
end
```

We can now add the implementation:

app/models/order.rb

```
class Order < ApplicationRecord
  # ...
  def set_total!
    self.total = products.map(&:price).sum
  end
end
```

Just before you run your tests, we need to update the `order` factory, just to make it more useful:

spec/factories/orders.rb

```
FactoryBot.define do
  factory :order do
    user { nil }
    total { 0.0 }
  end
end
```

We can now hook the `set_total!` method to a `before_validation` callback to make sure it has the correct total before is validated.

app/models/order.rb

```
class Order < ApplicationRecord
  before_validation :set_total!
  # ...
end
```

At this point, we are making sure the total is always present and bigger or equal to zero, meaning we can remove those validations and remove the specs. I'll wait. Our tests should be passing by now:

```
$ rspec spec/models/order_spec.rb
.........

Finished in 0.06807 seconds (files took 0.66165 seconds to load)
9 examples, 0 failures
```

This is now the moment to visit the `orders_controller_spec.rb` file and refactor some code. Currently we have something like:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...
  describe 'POST #create' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryBot.create :product
      product_2 = FactoryBot.create :product
      order_params = { total: 50, user_id: current_user.id,
product_ids: [product_1.id, product_2.id] }
      post :create, params: { user_id: current_user.id, order:
order_params }
    end

    it 'returns the just user order record' do
      expect(json_response[:id]).to be_present
    end

    it { expect(response.response_code).to eq(201) }
  end
end
```

If you run the tests now, they will pass, but first, let’s remove the `total` and `user_id` from the permitted params and avoid the mass-assignment. The `order_params` method should look like this:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...

  describe 'POST #create' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryBot.create :product
      product_2 = FactoryBot.create :product
      # changes heres
      order_params = { product_ids: [product_1.id, product_2.id]
}
      post :create, params: { user_id: current_user.id, order:
order_params }
    end

    it 'returns the just user order record' do
      expect(json_response[:id]).to be_present
    end

    it { expect(response.response_code).to eq(201) }
  end
end
```

If you run the tests now, they will pass, but first, let's remove the `total` and `user_id` from the permitted params and avoid the mass-assignment. The `order_params` method should look like this:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  # ...
  private

  def order_params
    params.require(:order).permit(product_ids: [])
  end
end
```

Your tests should still passing:

```
$ git commit -am "Adds the create method for the orders
controller"
```

Let’s commit the changes:

```
$ git commit -am "Adds the create method for the orders
controller"
```

Customizing the Order JSON output

Now that we built the necessary endpoints for the orders, we can customize the information we want to render on the JSON output for each order.

If you remember previous chapter we also use Active Model Serializers gem now. Let's generate a brand new serializer:

```
$ rails generate serializer order
```

Now let's open `order_serializer.rb` who should looks like:

app/serializers/order_serializer.rb

```ruby
class OrderSerializer < ActiveModel::Serializer
  attributes :id
end
```

We will add the products association and the total attribute to the order output, and to make sure everything is running smooth, we will some specs. In order to avoid duplication on tests, I'll just add one spec for the `show` and make sure the extra data is being rendered, this is because I'm using the same serializer every time an order object is being parsed to JSON, so in this case I would say it is just fine:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...
  describe 'GET #show' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token
      @order = FactoryGirl.create :order, user: current_user,
product_ids: [@product.id]
      get :show, params: { user_id: current_user.id, id:
@order.id }
    end

    it 'returns the user order record matching the id' do
      expect(json_response[:id]).to eql @order.id
    end

    it 'includes the total for the order' do
      expect(json_response[:total]).to eql @order.total.to_s
    end

    it 'includes the products on the order' do
      expect(json_response[:products]).to have(1).item
    end
    # ...
  end
  # ...
end
```

By now we should have failing tests. But they are easy to fix on the order serializer

app/serializers/order_serializer.rb

```ruby
class OrderSerializer < ActiveModel::Serializer
  attributes :id, :total
  has_many :products
end
```

And now all of our tests should be green:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
........

Finished in 0.22865 seconds (files took 0.70506 seconds to load)
8 examples, 0 failures
```

If you recall previous chapter we embedded the user into the product to retrieve some information. But in this case we always know the user because is actually the `current_user` so there is no point on adding it. It is not efficient. So let’s fix that by adding a new serializer:

```
$ rails g serializer order_product
```

We want to keep the products information consistent with the one we currently have. So we can just inherit behavior from it like so:

app/serializers/order_product_serializer.rb

```ruby
class OrderProductSerializer < OrderSerializer
end
```

This will keep rendered data on sync, and now to remove the embedded user we simply add the following method on the gem documentation. For more information visit ActiveModelSerializer:

app/serializers/order_product_serializer.rb

```ruby
class OrderProductSerializer < ProductSerializer
  def include_user?
    false
  end
end
```

After making this change we need to tell the `order_serializer` to use the serializer we just created by just passing an option to the `has_many` association on the `order_serializer`:

app/serializers/order_product_serializer.rb

```ruby
class OrderProductSerializer < ProductSerializer
  def include_user?
    false
  end
end
```

And our tests should still passing:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
........

Finished in 0.24024 seconds (files took 0.70072 seconds to load)
8 examples, 0 failures
```

Let’s commit this and move onto the next section:

```
$ git add .
$ git commit -m "Adds a custom order product serializer to
remove the user association"
```

Send order confirmation email

The last section for this chapter will be to sent a confirmation email for the user who just placed it. If you want to skip this and jump into the next chapter go ahead. This section is more like a warmup.

You may be familiar with email manipulation with Rails so I'll try to make this fast and simple. We first create the order_mailer:

```
$ rails generate mailer order_mailer
```

To make it easy to test the email, we will use a gem called email_spec, it includes a bunch of useful matchers for mailers, which makes it easy and fun.

So first let's add the gem to the Gemfile

Gemfile

```
# ...
group :test do
  gem 'rspec-collection_matchers', '~> 1.1'
  gem 'rspec-rails', '~> 3.8'
  gem "email_spec"
  gem 'shoulda-matchers'
end
# ...
```

Now run the bundle install command to install all the dependencies. I'll follow the documentation steps to setup the gem, you can do so on documentation. When you are done, your spec_helper.rb file should look like:

spec/rails_helper.rb

```ruby
require File.expand_path('../config/environment', __dir__)
ENV['RAILS_ENV'] ||= 'test'
# Prevent database truncation if the environment is production
abort('The Rails environment is running in production mode!') if
Rails.env.production?

require 'spec_helper'
require 'email_spec'
require 'email_spec/rspec'
require 'rspec/rails'
# ...
```

Now we can add some tests for the order mailer we created earlier:

```ruby
# ...
RSpec.describe OrderMailer, type: :mailer do
  include Rails.application.routes.url_helpers

  describe '.send_confirmation' do
    before(:all) do
      @user = FactoryBot.create :user
      @order = FactoryBot.create :order, user: @user
      @order_mailer = OrderMailer.send_confirmation(@order)
    end

    it 'should be set to be delivered to the user from the order
passed in' do
      expect(@order_mailer).to deliver_to(@user.email)
    end

    it 'should be set to be send from no-reply@marketplace.com'
do
      expect(@order_mailer).to deliver_from('no-
reply@marketplace.com')
    end

    it "should contain the user's message in the mail body" do
      expect(@order_mailer).to have_body_text(/Order: ##{
@order.id}/)
    end

    it 'should have the correct subject' do
      expect(@order_mailer).to have_subject(/Order
Confirmation/)
    end

    it 'should have the products count' do
      expect(@order_mailer).to have_body_text(/You ordered
#{@order.products.count} products:/)
    end
  end
end
```

I simply copied and pasted the one from the documentation and adapt it to our needs. We now have to make sure this tests pass. First we add the action on the order mailer:

app/mailers/order_mailer.rb

```ruby
class OrderMailer < ApplicationMailer
  default from: 'no-reply@marketplace.com'
  def send_confirmation(order)
    @order = order
    @user = @order.user
    mail to: @user.email, subject: 'Order Confirmation'
  end
end
```

After adding this code, we now have to add the corresponding views. It is a good practice to include a text version along with the html one.

```erb
<%# app/views/order_mailer/send_confirmation.txt.erb %>
Order: #<%= @order.id %>
You ordered <%= @order.products.count %> products:
<% @order.products.each do |product| %>
  <%= product.title %> - <%= number_to_currency product.price
%>
<% end %>
```

```erb
<!-- app/views/order_mailer/send_confirmation.html.erb -->
<h1>Order: #<%= @order.id %></h1>
<p>You ordered <%= @order.products.count %> products:</p>
<ul>
  <% @order.products.each do |product| %>
    <li><%= product.title %> - <%= number_to_currency product
.price %></li>
  <% end %>
</ul>
```

Now if we run the mailer specs, they should be all green:

```
$ rspec spec/mailers/order_mailer_spec.rb
.....

Finished in 0.24919 seconds (files took 0.75369 seconds to load)
5 examples, 0 failures
```

We just need to call the `send_confirmation` method into the create action on the orders controller:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  # ...
  def create
    order = current_user.orders.build(order_params)

    if order.save
      OrderMailer.send_confirmation(order).deliver
      render json: order, status: 201, location: [:api,
current_user, order]
    else
      render json: { errors: order.errors }, status: 422
    end
  end
  # ...
end
```

To make sure we did not break anything on the orders we can just run the specs from the orders controller:

```
$ rspec spec
.................................................................
.................................

Finished in 1.82 seconds (files took 0.78532 seconds to load)
98 examples, 0 failures
```

Let’s finish this section by committing this:

```
$ git add .
$ git commit -m "Adds order confirmation mailer"
```

Conclusion

Hey you made it! Give yourself an applause. I know it’s been a long way now, but you are almost done, believe me!.

On chapters to come we will keep working on the `Order` model to add some validations when placing an order, some scenarios are:

1. What happens when the products are not available?
2. Decrement the current product quantity when an order is placed

Next chapter will be short but is really important for the sanity of the app, so don’t skip it.

After chapter 9, we will focus on optimization, pagination and some other cool stuff that will definitely help you build a better app.

Improving orders

Back in previous chapter we extended our API to place orders and send a confirmation email to the user (just to improve the user experience). This chapter will take care of some validations on the order model, just to make sure it is placeable, just like:

1. Decrement the current product quantity when an order is placed
2. What happens when the products are not available?

We’ll probably need to update a little bit the JSON output for the orders but let’s not spoil things up.

So now that we have everything clear, we can get our hands dirty. You can clone the project up to this point with:

```
$ git clone https://github.com/madeindjs/market_place_api.git -
b chapter8
```

Let’s create a branch to start working:

```
$ git checkout -b chapter9
```

Decrementing the product quantity

On this first stop we will work on update the product quantity to make sure every order will deliver the actual product. Currently the `product` model doesn't have a `quantity` attribute, so let's do that:

```
$ rails generate migration add_quantity_to_products
quantity:integer
```

Wait, don't run the migrations just yet, we are making a small modification to it. As a good practice I like to add default values for the database just to make sure I don't mess things up with `null` values. This is a perfect case!

Your migration file should look like this:

db/migrate/20181227092237_add_quantity_to_products.rb

```ruby
class AddQuantityToProducts < ActiveRecord::Migration[5.2]
  def change
    add_column :products, :quantity, :integer, default: 0
  end
end
```

Now we can migrate database:

```
$ rake db:migrate
```

Now it is time to decrement the quantity for the `product` once an `order` is placed. Probably the first thing that comes to your mind is to take this to the `Order` model and this is a common mistake when working with *Many-to-Many* associations, we totally forget about the joining model which in this case is `Placement`.

The `Placement` is a better place to handle this as we have access to the order and the product, so we can easily in this case decrement the product stock.

Before we start implementing the code for the decrement, we have to change the way we handle the `order` creation as we now have to accept a quantity for each product. Remember we expecting we are expecting an array of product ids. I'm going to try to keep things simple and I will send an array of arrays where the first position of each inner array will be the product id and the second the quantity.

A quick example on this would be something like:

```
product_ids_and_quantities = [
  [1,4],
  [3,5]
]
```

This is going to be tricky so stay with me. Let's first build some unit tests:

spec/models/order_spec.rb

```
# ...
RSpec.describe Order, type: :model do
  # ...
  describe '#build_placements_with_product_ids_and_quantities'
do
    before(:each) do
      product_1 = FactoryBot.create :product, price: 100,
quantity: 5
      product_2 = FactoryBot.create :product, price: 85,
quantity: 10

      @product_ids_and_quantities = [[product_1.id, 2],
[product_2.id, 3]]
    end

    it 'builds 2 placements for the order' do
      expect { order
.build_placements_with_product_ids_and_quantities(@product_ids_
and_quantities) }.to change { order.placements.size }.from(0).
to(2)
    end
  end
end
```

Then into the implementation:

app/models/order.rb

```ruby
class Order < ApplicationRecord
  # ...

  # @param product_ids_and_quantities [Array] something like
this
  #        `[[product_1.id, 2], [product_2.id, 3]]`. Where first
item is
  #        `product_id` and the second is the quantity
  # @yield [Placement] placement build
  def build_placements_with_product_ids_and_quantities
(product_ids_and_quantities)
    product_ids_and_quantities.each do |product_id_and_quantity|
      id, quantity = product_id_and_quantity # [1,5]
      placement = placements.build(product_id: id)
      yield placement if block_given?
    end
  end
end
```

And if we run our tests, they should be all nice and green:

```
$ rspec spec/models/order_spec.rb
........

Finished in 0.33759 seconds (files took 3.54 seconds to load)
8 examples, 0 failures
```

The `build_placements_with_product_ids_and_quantities` will build the placement objects and once we trigger the `save` method for the order everything will be inserted into the database. One last step before committing this is to update the `orders_controller_spec` along with its implementation.

First we update the `orders_controller_spec` file:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  # ...
  describe 'POST #create' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryBot.create :product
      product_2 = FactoryBot.create :product
      order_params = {
        product_ids_and_quantities: [[product_1.id, 2],
[product_2.id, 3]]
      }
      post :create, params: { user_id: current_user.id, order:
order_params }
    end

    it 'embeds the two product objects related to the order' do
      expect(json_response[:products].size).to eql 2
    end
    # ...
  end
end
```

Then we need to update the `orders_controller`:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  # ...
  def create
    order = Order.create! user: current_user
    order.build_placements_with_product_ids_and_quantities
(params[:order][:product_ids_and_quantities])

    if order.save
      order.reload # need to reload associations
      OrderMailer.send_confirmation(order).deliver
      render json: order, status: 201, location: [:api,
current_user, order]
    else
      render json: { errors: order.errors }, status: 422
    end
  end
end
```

NOTE we removed the order_params method as we are handling the creation for the placements.*_

And last but not least, we need to update the products factory file, to assign a high quantity value, to at least have some products to play around in stock.

spec/factories/products.rb

```ruby
FactoryBot.define do
  factory :product do
    title { FFaker::Product.product_name }
    price { rand * 100 }
    published { false }
    user
    quantity { 5 }
  end
end
```

Let's commit this changes and keep moving:

```
$ git add .
$ git commit -m "Allows the order to be placed along with
product quantity"
```

Did you notice we are not saving the quantity for each product anywhere? There is no way to keep track of that. This can be fix really easy, by just adding a quantity attribute to the `Placement` model, so this way for each product we save its corresponding quantity. Let’s start by creating the migration:

```
$ rails generate migration add_quantity_to_placements
quantity:integer
```

As with the product quantity attribute migration we should add a default value equal to 0, remember this is optional but I do like this approach. The migration file should look like:

db/migrate/20181227104830_add_quantity_to_placements.rb

```ruby
class AddQuantityToPlacements < ActiveRecord::Migration[5.2]
  def change
    add_column :placements, :quantity, :integer, default: 0
  end
end
```

Then run the migrations:

```
$ rake db:migrate
```

Let’s document the `quantity` attribute through a unit test like so:

spec/models/placement_spec.rb

```ruby
# ...
RSpec.describe Placement, type: :model do
  # ...
  it { should respond_to :quantity }
  # ...
end
```

Now we just need to update the `build_placements_with_product_ids_and_quantities` to add the `quantity` for the placements:

app/models/order.rb

```ruby
class Order < ApplicationRecord
  # ...
  def build_placements_with_product_ids_and_quantities
(product_ids_and_quantities)
    product_ids_and_quantities.each do |product_id_and_quantity|
      product_id, quantity = product_id_and_quantity # [1,5]
      placements.build(product_id: product_id, quantity:
quantity)
    end
  end
end
```

Our `order_spec.rb` should be still green:

```
$ rspec spec/models/order_spec.rb
........

Finished in 0.09898 seconds (files took 0.74936 seconds to load)
8 examples, 0 failures
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Adds quantity to placements"
```

Extending the Placement model

It is time to update the product quantity once the order is saved, or more accurate once the placement is created. In order to achieve this we are going to add a method and then hook it up to an `after_create` callback.

Let's first update our `placement` factory to make more sense:

spec/factories/placements.rb

```
FactoryBot.define do
  factory :placement do
    order
    product
    quantity { 1 }
  end
end
```

And then we can simply add some specs:

spec/models/placement_spec.rb

```
# ...
RSpec.describe Placement, type: :model do
  # ...
  it { should respond_to :quantity }
  # ...
  describe '#decrement_product_quantity!' do
    it 'decreases the product quantity by the placement
quantity' do
      product = placement.product
      expect { placement.decrement_product_quantity! }.to change
{ product.quantity }.by(-placement.quantity)
    end
  end
end
```

The implementation is fairly easy as shown bellow:

app/models/placement.rb

```
class Placement < ApplicationRecord
  # ...
  after_create :decrement_product_quantity!

  def decrement_product_quantity!
    product.decrement!(:quantity, quantity)
  end
end
```

Validate quantity of products

As you remember from the beginning of the chapter we added the `quantity` attribute to the `Product` model. Now it is time to validate that there are enough products for the order to be placed.

In order to make things more interesting and spice things up we will do it through a custom validator, just to keep things cleaner and show you another cool technique to achieve custom validations.

For **custom validators** you can head to the documentation. Let's get our hands dirty.

First we need to add a `validators` directory under the `app` directory (Rails will pick it up for so we do not need to load it).

```
$ mkdir app/validators
$ touch app/validators/enough_products_validator.rb
```

Before we drop any line of code, we need to make sure to add a spec to the `Order` model to check if the order can be placed.

spec/models/order_spec.rb

```
# ...
RSpec.describe Order, type: :model do
  # ...
  describe "#valid?" do
    before do
      product_1 = FactoryBot.create :product, price: 100,
quantity: 5
      product_2 = FactoryBot.create :product, price: 85,
quantity: 10

      placement_1 = FactoryBot.build :placement, product:
product_1, quantity: 3
      placement_2 = FactoryBot.build :placement, product:
product_2, quantity: 15

      @order = FactoryBot.build :order
      @order.placements << placement_1
      @order.placements << placement_2
    end

    it "becomes invalid due to insufficient products" do
      expect(@order).to_not be_valid
    end
  end
end
```

As you can see on the spec, we first make sure that `placement_2` is trying to request more products than are available, so in this case the `order` is not supposed to be valid.

The test by now should be failing, let's turn it into green by adding the code for the validator:

app/validators/enough_products_validator.rb

```
class EnoughProductsValidator < ActiveModel::Validator
  def validate(record)
    record.placements.each do |placement|
      product = placement.product
      if placement.quantity > product.quantity
        record.errors[product.title.to_s] << "Is out of stock,
just #{product.quantity} left"
      end
    end
  end
end
```

I manage to add a message for each of the products that are out of stock, but you can handle it differently if you want. Now we just need to add the validator to the `Order` model like so:

app/models/order.rb

```
class Order < ApplicationRecord
  # ...
  validates_with EnoughProductsValidator
  # ...
end
```

And now if you run your tests, everything should be nice and green:

```
$ rspec spec/models/order_spec.rb
.........

Finished in 0.19136 seconds (files took 0.74912 seconds to load)
9 examples, 0 failures
```

Let’s commit the changes:

```
$ git add .
$ git commit -m "Adds validator for order with not enough
products on stock"
```

Updating the total

Did you realize that the total is being calculated incorrectly, because currently it is just adding the price for the products on the order regardless of the quantity requested. Let me add the code to clarify the problem:

Currently in the order model we have this method to calculate the amount to pay:

app/models/order.rb

```ruby
class Order < ApplicationRecord
  # ...
  def set_total!
    self.total = products.map(&:price).sum
  end
  # ...
end
```

Now instead of calculating the total by just adding the product prices, we need to multiply it by the quantity, so let’s update the spec first:

spec/models/order_spec.rb

```
# ...
RSpec.describe Order, type: :model do
  # ...
  describe '#set_total!' do
    before(:each) do
      product_1 = FactoryBot.create :product, price: 100
      product_2 = FactoryBot.create :product, price: 85

      placement_1 = FactoryBot.build :placement, product:
product_1, quantity: 3
      placement_2 = FactoryBot.build :placement, product:
product_2, quantity: 15

      @order = FactoryBot.build :order
      @order.placements << placement_1
      @order.placements << placement_2
    end

    it 'returns the total amount to pay for the products' do
      expect { @order.set_total! }.to change { @order.total.to_f
}.from(0).to(1575)
    end
  end
  # ...
end
```

And the implementation is fairly easy:

app/models/order.rb

```
class Order < ApplicationRecord
  # ...
  def set_total!
    self.total = 0.0
    placements.each do |placement|
      self.total += placement.product.price.to_f * placement
.quantity
    end
  end
  # ...
end
```

And the specs should be green:

```
$ rspec spec/models/order_spec.rb
.........

Finished in 0.20537 seconds (files took 0.74555 seconds to load)
9 examples, 0 failures
```

Let’s commit the changes and wrap up.

```
$ git commit -am "Updates the total calculation for order"
```

Conclusion

Oh you are here! Let me congratulate you. It’s been a long way since first chapter but you are one step closer. Actually the next chapter would be the last one, so try to take the most out of it.

The last chapter would be on how to optimize the API by using `pagination` and `caching`. So buckle up, it is going to be a bumpy ride.

Optimizations

Welcome to the last chapter of the book. It’s been a long way and you are only one step away from the end. Back in previous chapter we finish modeling the `Order` model and we could say that the project is done by now, but I want to cover some important details about optimization. The topics I’m going to cover in here will be:

- Setup more JSON:API specifications
- Pagination
- Caching

I will try to go as deep as I can trying to cover some common scenarios on this, and hopefully by the end of the chapter you’ll have enough knowledge to apply into some other scenarios.

If you start reading at this point, you’ll probably want the code to work on, you can clone it like so:

```
$ git clone https://github.com/madeindjs/market_place_api.git
-b chapter9
```

Let’s now create a branch to start working:

```
$ git checkout -b chapter10
```

Setup more JSON:API specifications

As I have been telling you since the beginning of this book, an important and difficult part of creating your API is deciding the output format. Fortunately, some organizations have already faced this type of problem and have established certain conventions.

One of the most applied conventions is most certainly JSON:API. This convention will allow us to approach pagination more serenely in the next section.

So documentation JSON:API gives us some rules to follow concerning JSON presentation.

So our **document** must follow theses rules:

- `data`: which must contains the data we send back
- `errors` which must contains a table of errors that have occurred
- `meta` which contains object meta

NOTE The `data` and `errors` keys must not be present at the same time and this makes sense since if an error occurs we should not be able to make data correct.

The content of the `data` key is also quite strict:

- it must have a `type` key that describes the type of the JSON model (if it is an article, a user, etc.)
- the properties of the objects must be placed in an `attributes` key and the underscore (`_`) are replaced by dashes (`-`)
- the links of the objects must be placed in a `relationships` key

This may cause us a lot of change since we have not implemented any of its rules. Don't worry, we have set up unit tests to ensure that there will be no regression. Let's start with doc updating them

Users

So let's start with users controller. Take a look at `show` action:

spec/controllers/api/v1/users_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'returns the information about a reporter on a hash' do
      expect(json_response[:email]).to eql @user.email
    end

    it 'has the product ids as an embedded object' do
      expect(json_response[:product_ids]).to eql []
    end
  end
  # ...
end
```

We just need to update emplacement of data. So the complete update file look like this:

spec/controllers/api/v1/users_controller_spec.rb

```
# ...
RSpec.describe Api::V1::UsersController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'returns the information about a reporter on a hash' do
      expect(json_response[:data][:attributes][:email]).to eql
@user.email
    end

    it 'has the product ids as an embedded object' do
      expect(json_response[:data][:attributes][:'product-ids'
]).to eql []
    end
  end

  describe 'POST #create' do
    context 'when is successfully created' do
      # ...
      it 'renders the json representation for the user record
just created' do
        expect(json_response[:data][:attributes][:email]).to eql
@user_attributes[:email]
      end
    end
    # ...
  end

  describe 'PUT/PATCH #update' do
    context 'when is successfully updated' do
      # ...
      it 'renders the json representation for the updated user'
do
        expect(json_response[:data][:attributes][:email]).to eql
'newmail@example.com'
      end
    end
    # ...
  end
  # ...
end
```

And that’s it. It display a lot of code but there are few changes.

User’s sessions

Only one test should be updated for user’s sessions: the one who get `auth_token`.

spec/controllers/api/v1/sessions_controller_spec.rb

```
# ...
RSpec.describe Api::V1::SessionsController, type: :controller do
  describe 'POST #create' do
    # ...
    context 'when the credentials are correct' do
      # ...
      it 'returns the user record corresponding to the given
credentials' do
        @user.reload
        expect(json_response[:data][:attributes][:'auth-token'
]).to eql @user.auth_token
      end
      # ...
    end
  # ...
  end
end
```

NOTE Remember that JSON:API specifications use dashes (`-`) instead of underscore (`_`)

Orders

There are one specificity for orders controller: we also get linked user. So to do so we need to use the `:relationships`. Apart from that, the principle remains the same:

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'returns the information about a reporter on a hash' do
      expect(json_response[:data][:attributes][:title]).to eql
@product.title
    end
    it 'has the user as a embedded object' do
      puts json_response.inspect
      expect(json_response[:data][:relationships][:user
][:attributes][:email]).to eql @product.user.email
    end
    # ...
  end

  describe 'GET #index' do
    # ...
    context 'when is not receiving any product_ids parameter' do
      # ...
      it 'returns 4 records from the database' do
        expect(json_response[:data]).to have(4).items
      end
      it 'returns the user object into each product' do
        json_response.each do |product_response|
          expect(product_response[:data][:relationships][:user
]).to be_present
        end
      end
      # ...
    end

    context 'when product_ids parameter is sent' do
      # ...
      it 'returns just the products that belong to the user' do
        json_response.each do |product_response|
          expect(product_response[:data][:relationships][:user
][:attributes][:email]).to eql @user.email
        end
      end
    end
  end

  describe 'POST #create' do
    context 'when is successfully created' do
      # ...
      it 'renders the json representation for the product record
just created' do
        expect(json_response[:data][:attributes][:title]).to eql
@product_attributes[:title]
      end
      # ...
    end
    # ...
  end

  describe 'PUT/PATCH #update' do
    # ...
    context 'when is successfully updated' do
      # ...
      it 'renders the json representation for the updated user'
do
        expect(json_response[:data][:attributes][:title]).to eql
'An expensive TV'
      end
      # ...
    end
    # ...
  end
  # ...
end
```

Product

Again, that's a lot of code, but in reality there's very little change.

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  describe 'GET #show' do
    # ...

    it 'returns the information about a reporter on a hash' do
      expect(json_response[:data][:attributes][:title]).to eql
@product.title
    end

    it 'has the user as a embedded object' do
      expect(json_response[:data][:relationships][:user
][:attributes][:email]).to eql @product.user.email
    end
  end

  describe 'GET #index' do
    # ...
    context 'when is not receiving any product_ids parameter' do
```

```
      # ...
      it 'returns 4 records from the database' do
        expect(json_response[:data]).to have(4).items
      end

      it 'returns the user object into each product' do
        json_response.each do |product_response|
          expect(product_response[:data][:relationships][:user
]).to be_present
        end
      end
    end

    context 'when product_ids parameter is sent' do
      # ...
      it 'returns just the products that belong to the user' do
        json_response.each do |product_response|
          expect(product_response[:data][:relationships][:user
][:attributes][:email]).to eql @user.email
        end
      end
    end
  end

  describe 'POST #create' do
    context 'when is successfully created' do
      # ...
      it 'renders the json representation for the product record
just created' do
        product_response = json_response
        expect(product_response[:data][:attributes][:title]).to
eql @product_attributes[:title]
      end
      # ...
    end

    context 'when is not created' do
      # ...
      it 'renders the json errors on why the user could not be
created' do
        product_response = json_response
        expect(product_response[:errors][:price]).to include 'is
not a number'
      end
      # ...
    end
  end

  describe 'PUT/PATCH #update' do
    # ...
    context 'when is successfully updated' do
      # ...
      it 'renders the json representation for the updated user'
do
        expect(json_response[:data][:attributes][:title]).to eql
'An expensive TV'
      end
      # ...
    end
    # ...
  end
  # ...
end
```

Implementation

From the beginning, in order to serialize our models, we used *Active Model Serializer*. Fortunately for us this library offers several **adapters**. The adapters are in a way JSON models to be applied to all our serializers. It's perfect.

The documentation of *Active Model Serializer* shows us a list of existing adapters. And if you see where I'm going with this there's one ready for the JSON:API model! To set it up, simply activate the adapt it by creating the following file:

config/initializers/activemodel_serializer.rb

```
ActiveModelSerializers.config.adapter = :json_api
```

We must also indicate the type of the serializer object. *Active Model Serializer* offers an all fate method for this: `type`. Implementation is therefore very easy:

app/serializers/order_serializer.rb

```ruby
class OrderSerializer < ActiveModel::Serializer
  type :order
  # ...
end
```

app/serializers/product_serializer.rb

```ruby
class ProductSerializer < ActiveModel::Serializer
  type :product
  # ...
end
```

app/serializers/user_serializer.rb

```ruby
class UserSerializer < ActiveModel::Serializer
  type :user
  # ...
end
```

And that's all! Now let's run **all** our tests to see if they pass:

```
$ rspec spec
...........F.F.F.................................................
......................................

Failures:

  1) Api::V1::ProductsController GET #show has the user as a
embedded object
     Failure/Error: expect
(json_response[:data][:relationships][:user][:attributes][:email
]).to eql @product.user.email
     ...

  2) Api::V1::ProductsController GET #index when is not
receiving any product_ids parameter returns the user object into
each product
     Failure/Error: expect
(product_response[:data][:relationships][:user]).to be_present
     ...

  3) Api::V1::ProductsController GET #index when product_ids
parameter is sent returns just the products that belong to the
user
     Failure/Error: expect
(product_response[:data][:relationships][:user][:attributes][:em
ail]).to eql @user.email
     ...

Finished in 1.35 seconds (files took 1.1 seconds to load)
103 examples, 3 failures
```

Argh…. All our tests pass but we see that the user associated with the product is not integrated in the answer. This is actually quite normal. The JSON:API documentation recommends using an `include` key rather than nesting models together.

So let's update our test:

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  describe 'GET #show' do
    # ...
    it 'has the user as a embedded object' do
      expect(json_response[:included].first[:attributes][
:email]).to eql @product.user.email
    end
  end

  describe 'GET #index' do
    # ...
    context 'when is not receiving any product_ids parameter' do
      # ...
      it 'returns the user object into each product' do
        expect(json_response[:included]).to be_present
      end
      # ...
    end

    context 'when product_ids parameter is sent' do
      # ...
      it 'returns just the products that belong to the user' do
        expect(json_response[:included].first[:id].to_i).to eql
@user.id
      end
    end
  end
  # ...
end
```

Here too implementation is very easy. We just need to add the `include` option directly into the controller’s action.

app/controllers/api/v1/products_controller.rb

```
class Api::V1::ProductsController < ApplicationController
  #...
  def index
    render json: Product.search(params), include: [:user]
  end

  def show
    render json: Product.find(params[:id]), include: [:user]
  end
  #...
end
```

Let's run all the tests again to make sure that our final implementation is correct:

```
$ rspec spec
.................................................................
......................................

Finished in 2.12 seconds (files took 1.4 seconds to load)
103 examples, 0 failures
```

And that's the job. Since we are happy with our work, let's do a commit:

```
$ git add .
$ git commit -m "Respect JSON:API response format"
```

Pagination

A very common strategy to optimize an array of records from the database, is to load just a few by paginating them and if you are familiar with this technique you know that in Rails is really easy to achieve it whether if you are using will_paginate or kaminari.

Then only tricky part in here is how are we suppose to handle the JSON output now, to give enough information to the client on how the array is paginated. If you recall first chapter I shared some resources on the practices I was going to be following in here. One of them was http://jsonapi.org/ which is a must-bookmark page.

If we read the format section we will reach a sub section called Top Level and in very few words they mention something about pagination:

> "meta": meta-information about a resource, such as pagination.

It is not very descriptive but at least we have a hint on what to look next about the pagination implementation, but don't worry that is exactly what we are going to do in here.

Let's start with the `products` list.

Products

We are going to start nice and easy by paginating the products list as we don't have any kind of access restriction which leads to easier testing.

First we need to add the kaminari gem to our `Gemfile`:

```
$ bundle add kaminari
```

Now we can go to the `index` action on the `products_controller` and add the pagination methods as pointed on the documentation:

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  # ...
  def index
    render json: Product.page(params[:page]).per(params
[:per_page]).search(params)
  end
  # ...
end
```

So far the only thing that changed is the query on the database to just limit the result by 25 per page which is the default. But we have not added any extra information to the JSON output.

We need to provide the pagination information on the `meta` tag in the following form:

```json
"meta": {
    "pagination": {
        "per_page": 25,
        "total_page": 6,
        "total_objects": 11
    }
}
```

Now that we have the final structure for the `meta` tag we just need to output it on the JSON response. Let's first add some specs:

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'GET #index' do
    before(:each) do
      4.times { FactoryBot.create :product }
      get :index
    end
    # ...
    it 'Have a meta pagination tag' do
      expect(json_response).to have_key(:meta)
      expect(json_response[:meta]).to have_key(:pagination)
      expect(json_response[:meta][:pagination]).to have_key
(:'per-page')
      expect(json_response[:meta][:pagination]).to have_key
(:'total-pages')
      expect(json_response[:meta][:pagination]).to have_key
(:'total-objects')
    end

    it { expect(response.response_code).to eq(200) }
  end
  # ...
end
```

The test we have just added should fail or, if we run the tests, two tests fail. It means we broke something else:

```
$ bundle exec rspec
spec/controllers/api/v1/products_controller_spec.rb
...F....F..........

Failures:

  1) Api::V1::ProductsController GET #index Have a meta
pagination tag
     ...

  2) Api::V1::ProductsController GET #index when product_ids
parameter is sent returns just the products that belong to the
user
     Failure/Error: total_pages: products.total_pages,

     NoMethodError:
       undefined method 'total_pages' for
#<Array:0x0000556f1ef85c68>
     # ./app/controllers/api/v1/products_controller.rb:12:in
'index'
     ...

Finished in 0.40801 seconds (files took 0.62979 seconds to load)
20 examples, 2 failures
```

The error is actually on the Product.search method. In fact Kaminari is waiting for a registration relationship instead of a table. It's very easy to repair:

app/models/product.rb

```ruby
class Product < ApplicationRecord
  # ...
  def self.search(params = {})
    products = params[:product_ids].present? ? Product.where(id:
params[:product_ids]) : Product.all
    # ...
  end
end
```

Have you noticed the change? Let me explain it to you. We simply replaced the Product.find method with Product.where using the product_ids parameters. The difference is that the where method returns an ActiveRecord::Relation and that's exactly what we need.

Now, if we restart the tests, the test we broke should now pass:

```
$ bundle exec rspec
spec/controllers/api/v1/products_controller_spec.rb
...F...............

Failures:

  1) Api::V1::ProductsController GET #index Have a meta
pagination tag
     ...

Finished in 0.41533 seconds (files took 0.5997 seconds to load)
20 examples, 1 failure
```

Now that we fixed that, let's add the pagination information, we need to do it on the `products_controller.rb` file:

app/controllers/api/v1/products_controller.rb

```
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: %i[create
update destroy]

  def index
    products = Product.search(params).page(params[:page]).per
(params[:per_page])
    render(
      json: products,
      include: [:user],
      meta: {
        pagination: {
          per_page: params[:per_page],
          total_pages: products.total_pages,
          total_objects: products.total_count
        }
      }
    )
  end
  # ...
end
```

Now if we run the specs, they should be all passing:

```
$ bundle exec rspec
spec/controllers/api/v1/products_controller_spec.rb
....................

Finished in 0.66813 seconds (files took 2.72 seconds to load)
20 examples, 0 failures
```

Now we have make a really amazing optimization for the products list endpoint. Now it is the client job to fetch the correct `page` with the correct `per_page` param for the records.

Let’s commit this changes and proceed with the orders list.

```
$ git add .
$ git commit -m "Adds pagination for the products index action
to optimize response"
```

Orders list

Now it’s time to do exactly the same for the `orders` list endpoint which should be really easy to implement. But first, let’s add some specs to the `orders_controller_spec.rb` file:

spec/controllers/api/v1/orders_controller_spec.rb

```ruby
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  describe 'GET #index' do
    before(:each) do
      current_user = FactoryBot.create :user
      api_authorization_header current_user.auth_token
      4.times { FactoryBot.create :order, user: current_user }
      get :index, params: { user_id: current_user.id }
    end

    it 'returns 4 order records from the user' do
      expect(json_response[:data]).to have(4).items
    end

    it 'Have a meta pagination tag' do
      expect(json_response).to have_key(:meta)
      expect(json_response[:meta]).to have_key(:pagination)
      expect(json_response[:meta][:pagination]).to have_key
(:'per-page')
      expect(json_response[:meta][:pagination]).to have_key
(:'total-pages')
      expect(json_response[:meta][:pagination]).to have_key
(:'total-objects')
    end

    it { expect(response.response_code).to eq(200) }
  end
  # ...
end
```

As you may already know, our tests are no longer passing:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
.F........

Failures:

  1) Api::V1::OrdersController GET #index Have a meta pagination
tag
     Failure/Error: expect(json_response).to have_key(:meta)
       expected #has_key?(:meta) to return true, got false
     # ./spec/controllers/api/v1/orders_controller_spec.rb:18:in
`block (3 levels) in <top (required)>'

Finished in 0.66262 seconds (files took 2.74 seconds to load)
10 examples, 1 failure
```

Let's turn the red into green:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  before_action :authenticate_with_token!

  def index
    orders = current_user.orders.page(params[:page]).per(params
[:per_page])
    render(
      json: orders,
      meta: {
        pagination: {
          per_page: params[:per_page],
          total_pages: orders.total_pages,
          total_objects: orders.total_count
        }
      }
    )
  end
  # ...
end
```

Now all the tests should be nice and green:

```
$ rspec spec/controllers/api/v1/orders_controller_spec.rb
..........

Finished in 0.35201 seconds (files took 0.9404 seconds to load)
10 examples, 0 failures
```

Let's place and commit, because a refactor is coming:

```
$ git commit -am "Adds pagination for orders index action"
```

Refactoring pagination

If you have followed this tutorial or if you are an experienced Rails developer, you probably like to keep things DRY. You may have noticed that the code we just wrote is duplicated. I think it's a good habit to clean up the code a little once the functionality is implemented.

We will first clean up these tests that we duplicated in the file `orders_controller_spec.rb` and `products_controller_spec.rb`:

```
it 'Have a meta pagination tag' do
  expect(json_response).to have_key(:meta)
  expect(json_response[:meta]).to have_key(:pagination)
  expect(json_response[:meta][:pagination]).to have_key(:'per-
page')
  expect(json_response[:meta][:pagination]).to have_key(:'total-
pages')
  expect(json_response[:meta][:pagination]).to have_key(:'total-
objects')
end
```

Let's add a `shared_examples` folder under the `spec/support/` directory:

```
$ mkdir spec/support/shared_examples
```

And on the `pagination.rb` file you can just add the following lines:

spec/support/shared_examples/pagination.rb

```
shared_examples 'paginated list' do
  it 'Have a meta pagination tag' do
    expect(json_response).to have_key(:meta)
    expect(json_response[:meta]).to have_key(:pagination)
    expect(json_response[:meta][:pagination]).to have_key(:'per-
page')
    expect(json_response[:meta][:pagination]).to have_key
(:'total-pages')
    expect(json_response[:meta][:pagination]).to have_key
(:'total-objects')
  end
end
```

This shared example can now be use as a substitute for the five tests on the `orders_controller_spec.rb` and `products_controller_spec.rb` files like so:

spec/controllers/api/v1/orders_controller_spec.rb

```
# ...
RSpec.describe Api::V1::OrdersController, type: :controller do
  describe 'GET #index' do
    # ...
    it_behaves_like 'paginated list'
    # ...
  end
end
```

spec/controllers/api/v1/products_controller_spec.rb

```
# ...
RSpec.describe Api::V1::ProductsController, type: :controller do
  # ...
  describe 'GET #index' do
    # ...
    it_behaves_like 'paginated list'
    # ...
  end
  # ...
end
```

And both specs should be passing.

```
$ rspec spec/controllers/api/v1/
.................................................

Finished in 0.96778 seconds (files took 1.59 seconds to load)
49 examples, 0 failures
```

Now that we made this simple refactor we can jump into the pagination implementation for the controllers and clean things up. If you recall the index action for both the products and orders controller they both have the same pagination format. So let's move this logic into a method called `pagination` under the `application_controller.rb` file. This way we can access it on any controller which needs pagination in the future.

app/controllers/application_controller.rb

```ruby
class ApplicationController < ActionController::API
  include Authenticable

  # @return [Hash]
  def pagination(paginated_array)
    {
      pagination: {
        per_page: params[:per_page],
        total_pages: paginated_array.total_pages,
        total_objects: paginated_array.total_count
      }
    }
  end
end
```

And now we can substitute the pagination hash on both controllers for the method, like so:

app/controllers/api/v1/orders_controller.rb

```ruby
class Api::V1::OrdersController < ApplicationController
  # ...
  def index
    orders = current_user.orders.page(params[:page]).per(params
[:per_page])
    render(
      json: orders,
      meta: pagination(orders)
    )
  end
  # ...
end
```

app/controllers/api/v1/products_controller.rb

```ruby
class Api::V1::ProductsController < ApplicationController
  # ...
  def index
    products = Product.search(params).page(params[:page]).per
(params[:per_page])
    render(
      json: products,
      include: [:user],
      meta: pagination(products)
    )
  end
  # ...
end
```

If you run the specs for each file they should be all nice and green:

```
$ rspec spec/controllers/api/v1/
.................................................

Finished in 0.92996 seconds (files took 0.95615 seconds to load)
49 examples, 0 failures
```

This would be a good time to *commit* the changes and move on to the next section on caching.

```
$ git add .
```

API Caching

There is currently an implementation to do caching with the gem active_model_serializers which is really easy to handle. Although in older versions of the gem, this implementation can change, it does the job.

If we make a request to the product list, we will notice that the response time takes about 174 milliseconds using cURL

```
$ curl -w 'Total: %{time_total}\n' -o /dev/null -s
http://api.marketplace.dev/products
Total: 0,174111
```

NOTE The -w option allows us to retrieve the time of the request, -o redirects the response to a file and -s hides the cURL display

By adding only one line to the ProductSerializer class, we will see a significant improvement in response time!

app/serializers/product_serializer.rb

```
class ProductSerializer < ActiveModel::Serializer
  # ...
  cache key: 'product', expires_in: 3.hours
end
```

app/serializers/order_serializer.rb

```
class OrderSerializer < ActiveModel::Serializer
  # ...
  cache key: 'order', expires_in: 3.hours
end
```

app/serializers/user_serializer.rb

```
class UserSerializer < ActiveModel::Serializer
  # ...
  cache key: 'user', expires_in: 3.hours
end
```

And that’s all! Let’s check for improvement:

```
$ curl -w 'Total: %{time_total}\n' -o /dev/null -s
http://api.marketplace.dev/products
Total: 0,021599
$ curl -w 'Total: %{time_total}\n' -o /dev/null -s
http://api.marketplace.dev/products
Total: 0,021979
```

So we went from 174 ms to 21 ms. The improvement is therefore enormous! Let’s commit our change a last time:

```
$ git commit -am "Adds caching for the serializers"
```

Conclusion

If you get to that point, it means you’re done with the book. Good work! You have just become a great API Rails developer, that’s for sure.

Thank you for bringing this great adventure with me, I hope you enjoyed the trip as much as I did. We should have a beer sometime.