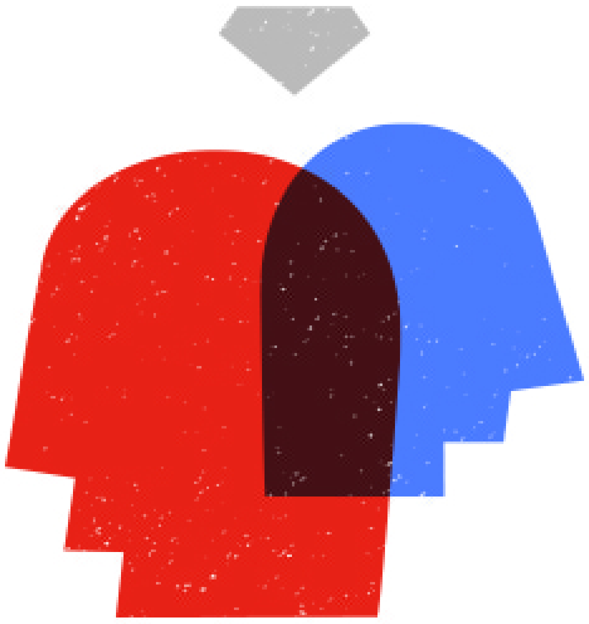


APIs on Rails

ABRAHAM KURI

APIs on Rails

Building REST APIs with Rails

Abraham Kuri

Contents

About the author

Abraham Kuri is a Rails developer with 5 years experience. His experience includes working as a freelancer building software products and more recently to collaborate into the open source community. He developed Furatto a front end framework built with Sass, Sabisu the next generation api explorer for your Rails app and have collaborated on other projects. He is a Computer Science graduate of ITESM and founded two companies in Mexico (Icalia Labs and Codeando Mexico).

Copyright and license

Chapter 1

Introduction

Welcome to APIs on Rails a tutorial on steroids on how to buid 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 http://apionrails.icalialabs.com; don't forget to update your offline version if that is the case.

The intention with this book it's not teach just how to build an API with Rails rather to teach you how to build *scalable and maintanable API with Rails*, which means taking your current Rails knowledge to the next level when on this approach. 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(Section 1.3) the project we will start tracking it with Git (Section 1.4).

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 decising wheter to use JSON or XML as the response format. From Chapter 3 to Chapter 8 we'll get our hands dirty 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 (Chapter 11) 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

1.1 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 shout me tweet or email me. I'm always willing to know you guys!

1.2 Getting started

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 as an attempt to make this easier and keep you motivated we will be using a bash script I manage put together called Kaishi, it includes all the necessary tools (Box 1.1) and more to setup your development environment, it currently only works for Mac OS:

Box 1.1. Kaishi development tools

- oh-my-zsh as your default shell
- Homebrew for managing packages
- Git for version controlling
- Postgresql as the database manager
- Vim for text editing
- ImageMagick for images processing
- Rbenv for managing the ruby environment
- Bundler gem
- Foreman for running apps
- Rails gem for creating any rails app
- Heroku toolbelt to interact with the Heroku API
- RailsAppCustomGenerator for initializing any Rails app with Icalia's flavor
- Pow to run local apps locally like a superhero

1.2.1 Development environments

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.

Now for those who are more like me, I can tell you that there are a lot of options out there which you can customize via plugins and more.

- **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 Chrome immediately, but some other developers may say Firefox 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.

A note on tools

All right, I understand that you may not want to include every single package that comes with kaishi, and that is fair, or maybe you already have some tools installed, well I'll describe you how to install the bare bones you need to get started:

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:

```
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

- **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.

- **Mac OS**:

```
$ brew install git
```

- **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 (1.8) if you are on Mac OS, to see which version you have, just type:

```
$ ruby -v
```

Rails 4 requires you to install version 1.9 or higher, and in order to accomplish this I recommend you to start using Ruby Version Manager (RVM) or rbenv, any of these will allow you to install multiple versions of **`ruby`**. I recently changed from RVM to rbenv and it's great, so any of these two options you choose is fine. On this tutorial we'll be using **`rbenv`**.

A note for Mac OS: if you are using Mac just keep in mind you have to have installed the Command Line Tools for Xcode.

Mac OS:

To get started with the ruby installation, type in:

```
$ rbenv install 2.1.2
```

Next you have to set up the just installed version of ruby as the default one:

```
$ rbenv global 2.1.2
$ rbenv rehash
```

The rehash command is supposed to run everytime you install a new ruby version or a gem. Seems like a lot? check out rbenv-gem-rehash brew formula to mitigate this.

For more information about customization or other types of installation checkout out the project documentation.

Linux:

The first steo is to setup some dependencies for Ruby:

```
$ sudo apt-get update
$ sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev \
                         libreadline-dev libyaml-dev libsqlite3-dev sqlite3 \
                         libxml2-dev libxslt1-dev libcurl4-openssl-dev \
                         python-software-properties
```

Next it is time to install ruby:

```
$ cd
$ git clone git://github.com/sstephenson/rbenv.git .rbenv
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.profile
$ echo 'eval "$(rbenv init -)"' >> ~/.profile
$ exec $SHELL

$ git clone git://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build
$ echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.profile
$ exec $SHELL

$ rbenv install 2.1.2
$ rbenv global 2.1.2
```

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 4.0
```

Check for everything to be running nice and smooth:

```
$ rails -v
Rails 4.0.5
```

Databases

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
```

1.3 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 (Listing 1.1):

Heads up: Be aware that we'll be using Rspec as the testing suite, so just make sure you include the *-T* option when creating the rails application.

Listing 1.1: Initializing the project with rails new.

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

As you may guess, the commands above(Listing 1.1) 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.

1.3.1 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:

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 like the one shown on Figure 1.1.

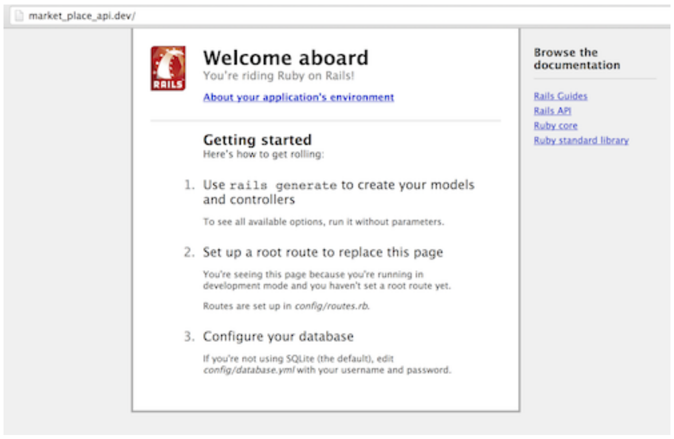
Installing Prax

For linux users only, I extracted the instructions from the official documentation, so for any further documentation you should refer to the README file on the github repository.

It is recommended that you clone the repository under the `/opt` directory and then run the installer which will set the port forwarding script and NSSwitch extension.

```
$ sudo git clone git://github.com/ysbaddaden/prax.git /opt/prax

$ cd /opt/prax/
$ ./bin/prax install
```

Then we just need to link the apps:



market_place_api.dev/

Welcome aboard

You're riding Ruby on Rails!

About your application's environment

Getting started

Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a root route to replace this page

   You're seeing this page because you're running in development mode and you haven't set a root route yet.

   Routes are set up in *config/routes.rb*.

3. Configure your database

   If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides

Rails API

Ruby core

Ruby standard library

Figure 1.1: http://market_place_api.dev/

```
$ 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
```

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 Figure 1.1.

1.3.2 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, is well maintained and the documentation is amazing.

So your `Gemfile` should look like this (Listing 1.2) after adding the `active_model_serializers` gem:

Listing 1.2: The default Gemfile with the serializers gem.

```ruby
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.2'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'

#Api gems
gem 'active_model_serializers'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end
```

Notice that I remove the `jbuilder` and `turbolinks` gems, as 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 (Listing 1.3):

Listing 1.3: The updated Gemfile for different groups.

```ruby
...
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 about deployment: 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. See Section 1.3.1

> 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
Fetching source index for https://rubygems.org/
.
.
.
```

After the command finish its execution, it is time to start tracking the project with git (Section 1.4)

1.4 Version Control

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 https://github.com/kurenn/market_place_api.

By this point I'll asume 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 "mvim -f"
```

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 (Listing 1.4):

Listing 1.4: The modified version of the .gitignore file

```
/.bundle

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

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

# Extra files to ignore
doc/
*.swp
*~
.DS_Store
```

After modifiying 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 "Initializes the project"
```

Good practice: I have encounter that commiting 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:kurenn/market_place_api.git
```

then:

```
$ git push -u origin master
Counting objects: 58, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (47/47), done.
Writing objects: 100% (58/58), 13.84 KiB | 0 bytes/s, done.
Total 58 (delta 2), reused 0 (delta 0)
To git@github.com:kurenn/market_place_api.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

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.

1.5 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. If you feel like want to share how are you doing with the tutorial, I'll be happy to read it, a nice example is shown below:

I just finished the first chapter of Api on Rails tutorial by @kurenn!

So let's get our hands dirty and start typing some code!

Chapter 2

The API

In this section I'll outline the application, by now you should have the bare bones of the application, as shown in Section 1.3, 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/kurenn/market_place_api.git
$ git checkout -b chapter1 2d95d071ebcd97f624175a428c21ef6797fe881a
```

And as a quick recap, we really just update the `Gemfile` to add the `active_model_ser` gem, see Listing 1.2 for more information.

2.1 Planning the application

As we want to go simple with the application, it consists on 5 models (Figure 2.1). 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

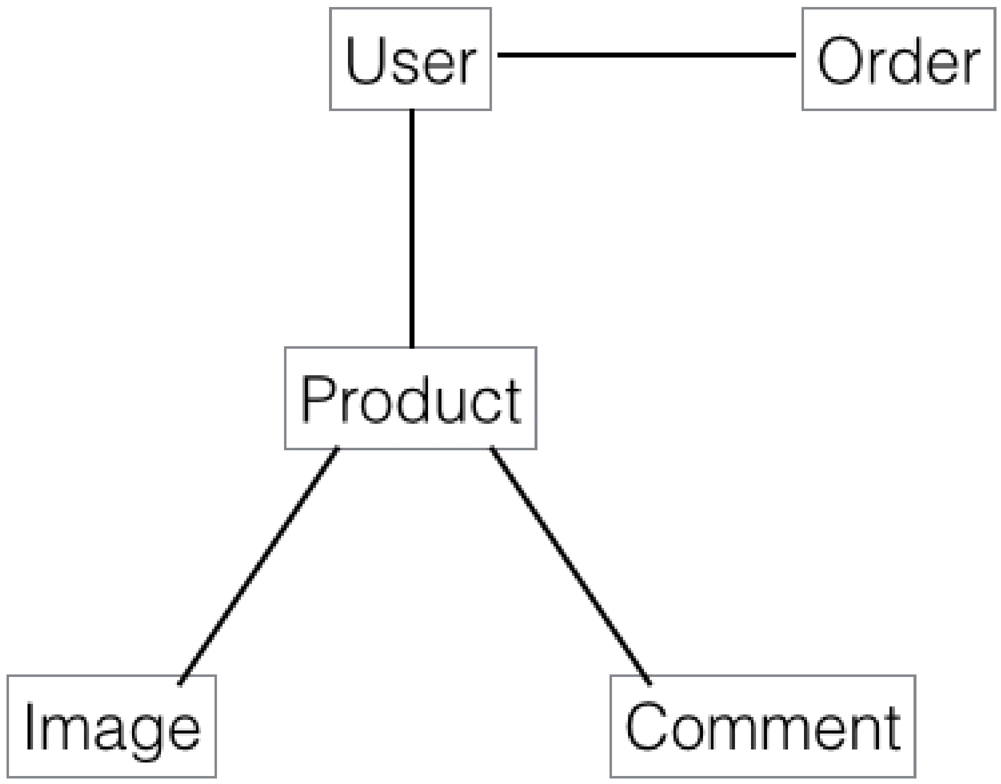


Figure 2.1

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 add on if you are using chrome, but we won't be using that anyway, as probably not every developer uses the Google browser. Instead of that we will be using a gem I built called **`sabisu_rails`** which is a powerful postman-like engine client to explore your Rails application api's. We will cover the gem integration on Section 3.3

2.2 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(Listing 2.1), CORBA and RPC. A REST call is* *simply a GET HTTP request to the server.*

Listing 2.1: SOAP call example

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

And in REST you may call a URL with an especific HTTP request, in this case with a `GET` request (Listing 2.2)

Listing 2.2: REST call example

```
http://domain.com/resources_name/uri_pattern
```

RESTful APIs must follow at least 3 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.

Resource	GET
`http://example.com/resources`	**Reads** the resource or resources defined by the U

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.

2.2.1 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` (Listing 2.3), as we are just going to set the `constraints`, the `base_uri` and the default response `format` for each request.

Listing 2.3: Default routes.rb file

```
MarketPlaceApi::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"
[setting-api f5e98f7] Removes comments from the routes file
 1 file changed, 3 insertions(+), 56 deletions(-)
 rewrite config/routes.rb (97%)
```

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. (Listing 2.4)

Listing 2.4: Commands to create the api from the terminal

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

We then add that namespace into our `routes.rb` file (Listing 2.5):

Listing 2.5: Routes with a namespace defined

```
MarketPlaceApi::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 can handle up to 21 different media types, you can list them by accessing the SET class under de Mime module:

```
$ rails c
Loading development environment (Rails 4.0.2)
2.1.0 :001 > Mime::SET.collect(&:to_s)
=> ["text/html", "text/plain", "text/javascript", "text/css", "text/calendar",
    "text/csv", "image/png", "image/jpeg", "image/gif", "image/bmp",
    "image/tiff", "video/mpeg", "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"]
```

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 (Listing 2.6):

Listing 2.6: Routes with a namespace and default format defined

```
MarketPlaceApi::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 achieve next is how 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 scalate the application to a DNS level. So how do we achieve that? (Listing 2.7)

Listing 2.7: Routes with a namespace, default format and subdomain defined

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

Notice the changes?, we didn't just add a **`constraints`** hash to specify the subdomain, but we also add the **`path`** option, and set it a *backslash*. This is telling Rails to set the starting path for each request to be root in relation to the subdomain(Box 2.1), achieving what we are looking for.

Box 2.1. Common api patterns

You can find many approaches to set up the base_uri when building an api following different patterns, assuming we are versioning our api (Section 2.2.2):

URI Pattern	Description
`api.example.com/`	I my opinion this is the way to go, gives you a better interface an
`example.com/api/`	This pattern is very common, and it is actually a good way to go
`example.com/api/v1`	This seems like a good idea, by setting the version of the api thro

Don't worry about versioning right now, I'll walk through it on (section 2.2.2)

Time to commit:

```
$ git add config/routes.rb
$ git commit -m "Set the routes contraints for the api"
[setting-api a5f2e0d] Set the routes contraints for the api
 1 file changed, 4 insertions(+), 1 deletion(-)
```

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

2.2.2 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:

```
MarketPlaceApi::Application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json },
                              constraints: { subdomain: 'api' }, path: '/'  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.

> Your API versioning is wrong, which is why I decided to do it 3 different wrong ways –Troy Hunt

In order to set the version for the api, we first need to add another directory under the `api` we created on (Section 2.2.1):

```
$ 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 (Listing 2.8)

Listing 2.8: Routes version specified

```
MarketPlaceApi::Application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json },
                              constraints: { subdomain: 'api' }, path: '/'  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
```

2.2.3 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(Box 2.2)

Box 2.2. Request headers description

HTTP header fields are components of the message header of requests and responses in the Hypertext Transfer Protocol (HTTP). They define the operating parameters of an HTTP transaction.

A common list of used headers is presented below:

Header Name	Description
`Accept`	Content-Types that are acceptable for the response
`Authorization`	Authentication credentials for HTTP authentication
`Content-Type`	The MIME type of the body of the request (used with POST and PUT r
`Origin`	Initiates a request for cross-origin resource sharing (asks server for an ‘
`User-Agent`	The user agent string of the user agent

The information above was extracted from http://en.wikipedia.org/wiki/List_of_HTTP_header_fields , I just managed to put the most common.

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. (Listing 2.11).

First we need to add our testing suite, which in our case is going to be Rspec (Listing 2.9):

Listing 2.9: Gemfile with the testing suite

```
.
.
.
group :test do
  gem "rspec-rails", "~> 2.14"
  gem "factory_girl_rails"
  gem 'ffaker'
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 g rspec:install
```

Listing 2.10: Configuring our test suite (*/config/application.rb*)

```
# don't generate RSpec tests for views and helpers
  config.generators do |g|
    g.test_framework :rspec, fixture: true
    g.fixture_replacement :factory_girl, 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)
```

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:

Listing 2.11: Api constraints spec *lib/spec/api_constraints_spec.rb*

```
require 'spec_helper'

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"})
      api_constraints_v1.matches?(request).should be_true
    end

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

Let me walk you through the code. We are initialising the class with an options hash, which 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(Listing 2.12)

Listing 2.12: Api constraints class *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.(Listing 2.13):

Listing 2.13: Routes file *config/routes.rb*

```ruby
require 'api_constraints'

MarketPlaceApi::Application.routes.draw do
  # Api definition
  namespace :api, defaults: { format: :json },
                              constraints: { subdomain: 'api' }, path: '/'  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 0.00238 seconds
2 examples, 0 failures

Randomized with seed 45035
```

2.3 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

Wanna tweet about your accomplishment?:

I just finished the second chapter of Api on Rails tutorial by @kurenn!

Chapter 3

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 (Chapter 5) 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/kurenn/market_place_api.git -b chapter2
```

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 (Box 3.1), which offers a great way to integrate not just basic authentication, but many other modules for further use.

Box 3.1. Devise for authentication

Flexible authentication solution for Rails with Warden. - Plataformatec

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.

3.1 User model

We need to first add the **devise** gem into the **Gemfile** (Listing 3.1):

Listing 3.1: Gemfile with devise

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.2'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'

#Api gems
gem 'active_model_serializers'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

group :development do
  gem 'sqlite3'
end

group :test do
  gem "rspec-rails"
  gem "factory_girl_rails"
  gem 'ffaker'
end

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/20140622003340_devise_create_users.rb
      create     app/models/user.rb
      invoke     rspec
      create       spec/models/user_spec.rb
      invoke       factory_girl
      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(Listing 3.2). This will help us to easily create test users and facilitate our tests writing.

Listing 3.2: User factory (*spec/factories/users.rb*)

```ruby
FactoryGirl.define do
  factory :user do
  end
end
```

Next we migrate the database and prepare the test database.

```
$ rake db:migrate
==  DeviseCreateUsers: migrating ==============================================
-- create_table(:users)
   -> 0.0031s
-- add_index(:users, :email, {:unique=>true})
```

```
   -> 0.0010s
-- add_index(:users, :reset_password_token, {:unique=>true})
   -> 0.0004s
==  DeviseCreateUsers: migrated (0.0047s) =====================================
```

```
$ 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"
[chapter3 be4e03f] Adds devise user model
 10 files changed, 423 insertions(+), 1 deletion(-)
 create mode 100644 app/models/user.rb
 create mode 100644 config/initializers/devise.rb
 create mode 100644 config/locales/devise.en.yml
 create mode 100644 db/migrate/20140622003340_devise_create_users.rb
 create mode 100644 db/schema.rb
 create mode 100644 spec/factories/users.rb
 create mode 100644 spec/models/user_spec.rb
```

3.1.1 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(Listing 3.4). Also for convenience we will modify the `users` factory file to add the corresponding attributes (Listing 3.3).

Listing 3.3: User factory file with attributes (*spec/factories/users.rb*)

```
FactoryGirl.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 (Listing 3.4).

Listing 3.4: Testing for devise provided attributes (*spec/models/user_spec.rb*)

```
require 'spec_helper'

describe User do
  before { @user = FactoryGirl.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
```

Our tests should pass.

```
....

Finished in 0.03747 seconds
4 examples, 0 failures

Randomized with seed 31597
```

That was easy, we should probably commit this changes:

```
$ git add .
$ git commit -am 'Adds user firsts specs'
[chapter3 80b4cb6] Adds user firsts specs
 2 files changed, 12 insertions(+), 1 deletion(-)
```

3.1.2 Improving validation tests

As you may notice in Section 3.1.1 we did not add tests for validations, even though devise already add some on the `user` model. We could write some tests which could look like this in case of validating email presence (Listing 3.5):

Listing 3.5: Validation tests without shoulda-matchers (*spec/models/user_spec.rb*)

```
describe "when email is not present" do
  before { @user.email = " " }
  it { should_not be_valid }
end
```

But this can be improved with a `gem` called `shoulda-matchers`, let's add it to our `Gemfile`:

```
.
.
.
group :test do
  gem "rspec-rails"
  gem "factory_girl_rails"
  gem 'ffaker'
  gem "shoulda-matchers"
end
.
.
.
```

As usual before adding a gem, run the `bundle` command to install it. Shoulda-matchers help us refactor our tests(Listing 3.5) like so:

```
it { should validate_presence_of(:email) }
```

Let's finish up the `user` specs and commit the changes:

```
it { should validate_presence_of(:email) }
it { should validate_uniqueness_of(:email) }
it { should validate_confirmation_of(:password) }
it { should allow_value('example@domain.com').for(:email) }
```

```
$ git add .
$ git commit -m "Adds shoulda matchers for spec refactors"
[chapter3 9f8c6c1] Adds shoulda matchers for spec refactors
 3 files changed, 10 insertions(+)
```

3.2 Building users endpoints

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`(Listing 3.6). 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 (Box 3.2).

Box 3.2. 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 recognises these five classes - Wikipedia

A common list of used http codes is presented below:

HTTP Code	Description
`200`	Standard response for successful HTTP requests
`201`	The request has been fulfilled and resulted in a new resource being created.
`204`	The server successfully processed the request, but is not returning any conten
`400`	The request cannot be fulfilled due to bad syntax.
`401`	Similar to 403 Forbidden, but specifically for use when authentication is requ
`404`	The requested resource could not be found but may be available again in the f
`500`	A generic error message, given when an unexpected condition was encounter

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

To keep our code nicely organised, we will create some directories under the controller specs directory in order to be consistent with our current setup (Listing 2.4). 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:

```ruby
require 'spec_helper'

describe Api::V1::UsersController do

end
```

Now with tests added your file should look like:

Listing 3.6: Users controller spec file (*spec/controllers/api/v1/users_controller_spec.rb*)

```ruby
require 'spec_helper'

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

  describe "GET #show" do
    before(:each) do
      @user = FactoryGirl.create :user
      get :show, 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 { should respond_with 200 }
  end
end
```

So far, the tests look good, we just need to add the implementation(Listing 3.7), it is extremely simple:

Listing 3.7: Users controller file (*app/controllers/api/v1/users_controller.rb*)

```ruby
class Api::V1::UsersController < ApplicationController
  respond_to :json

  def show
    respond_with User.find(params[:id])
  end
end
```

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

```
FF

Failures:
  1) Api::V1::UsersController GET #show
     Failure/Error: get :show, id: @user.id, format: :json
     ActionController::UrlGenerationError:
       ...

  2) Api::V1::UsersController GET #show returns the information about a...
     Failure/Error: get :show, id: @user.id, format: :json
     ActionController::UrlGenerationError:
       ...

Finished in 0.0552 seconds
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:

```
require 'api_constraints'

MarketPlaceApi::Application.routes.draw do
  devise_for :users
  # Api definition
  namespace :api, defaults: { format: :json },
                              constraints: { subdomain: 'api' }, path: '/'  do
    scope module: :v1,
              constraints: ApiConstraints.new(version: 1, default: true) do
      # We are going to list our resources here
      resources :users, :only => [:show]
    end
  end
end
```

If you try again `bundle exec rspec spec/controllers` you will see your all your tests passing.

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"
```

3.2.1 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, which comes built-in on almost any Linux distribution and of course on your Mac OSX. So let's try it out:

Remember our base uri is `api.market_place_api.dev` (Section 2.2.1).

```
$ 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, we don't have a user with `id` 1, let's create it first through the terminal:

```
$ rails console
Loading development environment (Rails 4.0.2)
2.1.0 :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":"a@a.com","created_at":"2014-06-23T04:24:42.031Z",...
```

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 (Listing 3.8):

Listing 3.8: Application controller for api responses (*app/controllers/application_controller.rb*)

```
class ApplicationController < ActionController::Base
  # 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"
```

3.2.2 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 (Box 3.2) 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 (Listing 3.9):

Listing 3.9: Users controller spec file with create tests (*spec/controllers/api/v1/users_controller_spec.rb*)

```
describe "POST #create" do

  context "when is successfully created" do
    before(:each) do
      @user_attributes = FactoryGirl.attributes_for :user
      post :create, { 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 { should respond_with 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, { 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 { should respond_with 422 }
    end
  end
```

There is a lot of code up there(Listing 3.9) 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:

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

Time to implement some code and make our tests pass (Listing 3.10)

Listing 3.10: Users controller file with create action (*app/controllers/api/v1/users_controller.rb*)

```
.
.
.
  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
.
.
.
```

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

```
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
      # We are going to list our resources here
      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 0.12152 seconds
7 examples, 0 failures

Randomized with seed 24824
```

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

```
$ git add .
$ git commit -m "Adds the user create endpoint"
[chapter3 9222e75] Adds the user create endpoint
 3 files changed, 52 insertions(+), 1 deletion(-)
```

3.2.3 Updating users

The pattern for `updating` users is very similar as `creating` new ones (Section 3.2.2). If you are an experienced rails developer you may already know the differences between these two actions, and the implications:

- The `update` action responds to a PUT/PATCH request (Section 2.2).
- 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 Chapter 5

As usual we start by writing our tests (Listing 3.11):

Listing 3.11: Users controller spec file with update tests (*spec/controllers/api/v1/users_controller_spec.rb*)

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

    context "when is successfully updated" do
      before(:each) do
        @user = FactoryGirl.create :user
        patch :update, { 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 { should respond_with 200 }
    end

    context "when is not created" do
      before(:each) do
        @user = FactoryGirl.create :user
        patch :update, { 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 whye 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 { should respond_with 422 }
    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 Chapter 4

First we add the action the **`routes.rb`** file

```
.
.
.
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  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*):

```
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
```

If we run our tests, we should now have all of our tests passing.
We commit the changes as we added a bunch of working code:

```
$ git add .
$ git commit -m "Adds update action the users controller"
[chapter3 d4422ff] Adds update action the users controller
 3 files changed, 48 insertions(+), 1 deletion(-)
```

3.2.4 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 (Listing 3.12):

Listing 3.12: Users controller spec file with destroy tests (*spec/controllers/api/v1/users_controller_spec.rb*)

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

  it { should respond_with 204 }

end
```

As you can see the spec is very simple, as we only respond with a status of `204` which stands for `No Content`, meaning 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 nothing 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:

```
def destroy
  user = User.find(params[:id])
  user.destroy
  head 204
end
```

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

```
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  resources :users, :only => [:show, :create, :update, :destroy]
end
```

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

```
.............

Finished in 0.25011 seconds
13 examples, 0 failures

Randomized with seed 16839
```

Remember after making some changes to our code, it is good practice to commit them so that we keep our history very atomic.

```
$ git add .
$ git commit -m "Adds destroy action to the users controller"
[chapter3 2034ddf] Adds destroy action to the users controller
 3 files changed, 17 insertions(+), 1 deletion(-)
```

3.3 Integrating Sabisu

As shown on Section 3.2.1 we play around a bit with `curl` to test our enpoints, or at least the `index` action. Well, even is a good starting point to see the api on action, as the application grows it becomes harder to maintain and visualize, with many endpoints all over the place and quite large `json` outputs, things will get nasty in the short term.

At Icalia Labs we are always looking for tools which make our life easy, and when working with an api, a fine solution we have found is PostMan, which is a `REST` client for inspecting any api and even better comes as a chrome add on. One thing we encounter with this app though, was when sharing the application among other developers, we had to go part by part and explain them what was going on around the enpoints, which seems a bit of waste of time. That's why I put on together a `gem` which mounts on to your rails api and maps your resources for placing requests (Box 3.3).

Box 3.3. Sabisu gem

> *Sabisu is a powerful postman-like engine client to explore your Rails application api. It's still under heavy development, but it is quite impressive and beautiful.* – Abraham Kuri

This tool was developed with the intention to reduce the learning curve when integrating new teammates to the project, or simply because is will just run on any browser and offers, almost the same support as postman.

It uses well supported dependencies such as:

- Compass
- Furatto
- Simple Form
- Font Awesome

Even better you can customise to never add the attributes for creating a new record, believe me, I have worked with models with up to 20 attributes and I've had to add them manually, it really takes a lot of time.

You will know what I mean as we move forward with the **gem** integration.

Following the documentation from the repo:

We add the necessary gems to the **Gemfile**

```
gem 'sabisu_rails', github: "IcaliaLabs/sabisu-rails"
gem 'compass-rails'
gem 'furatto'
gem 'font-awesome-rails'
gem 'simple_form'
```

Then run the **bundle** command to install them.

```
$ bundle install
```

After installing the gems via **bundler**, we run the **simple_form** and **sabisu** generators:

```
$ rails generate simple_form:install
$ rails generate sabisu_rails:install
```

The commands above will generate two initialiser files, we only care about the one related to **sabisu** located on *config/initializers/sabisu_rails.rb*.

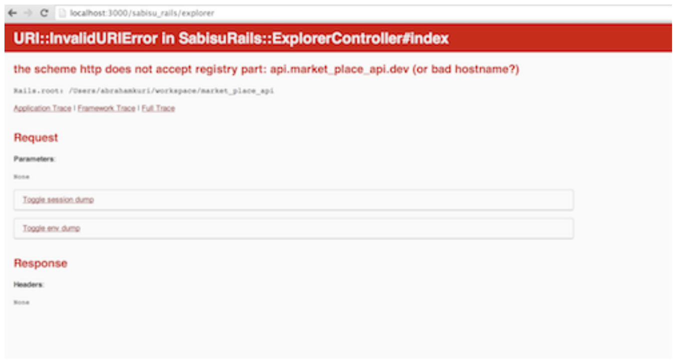
If everything went smooth we just have to customise the **sabisu** engine (Listing 3.13):

Listing 3.13: Sabisu initialiser customization (*config/initializers/sabisu_rails.rb*)

```
# Use this module to configure the sabisu available options

SabisuRails.setup do |config|

  # Base uri for posting the
  config.base_api_uri = 'api.market_place_api.dev'

  # Ignored attributes for building the forms
  # config.ignored_attributes = %w{ created_at updated_at id }

  # HTTP methods
  # config.http_methods = %w{ GET POST PUT DELETE PATCH }

  # Headers to include on each request
  #
  # You can configure the api headers fairly easy by just adding the correct headers
   config.api_headers = { "Accept" => "application/vnd.marketplace.v1" }
  #
  # config.api_headers = {}

  # Layout configuration
  # config.layout = "sabisu"

  # Resources on the api
   config.resources = [:users]

  # Default resource
   config.default_resource = :users

  # Application name
  # mattr_accessor :app_name
  # @@app_name = Rails.application.class.parent_name

  # Authentication
  # mattr_accessor :authentication_username
  # @@authentication_username = "admin"

  # mattr_accessor :authentication_password
  # @@authentication_password = "sekret"

end
```

You can customize it further to meet other requirements, but for now this just works fine. Now let's try the explorer now, by running `rails server` on the console and open up the browser with the following url http://localhost:3000/sabisu_rails (Figure 3.1)

If you don't see the error message and instead a window for authentication pops up, the credentials to access are `admin` for the username and `sekret`



localhost:3000/sabisu_rails/explorer

URI::InvalidURIError in SabisuRails::ExplorerController#index

the scheme http does not accept registry part: api.market_place_api.dev (or bad hostname?)

```
Rails.root: /Users/abrahamkuri/workspace/market_place_api
```

Application Trace | Framework Trace | Full Trace

Request

Parameters:

```
None
```

Toggle session dump

Toggle env dump

Response

Headers:

```
None
```

Figure 3.1: http://api.market_place_api.dev/sabisu_rails/explorer

for the password. These settings can be customized under the **`sabisu_rails`** config file under **`config/initializers`**.

But wait a minute there is an error over there. this is something I have found when using **`pow`** and httparty. For some reason URIs with underscore cannot be parsed correctly so we get:

- *the scheme http does not accept registry part: api.market_place_api.dev (or bad hostname?)* -

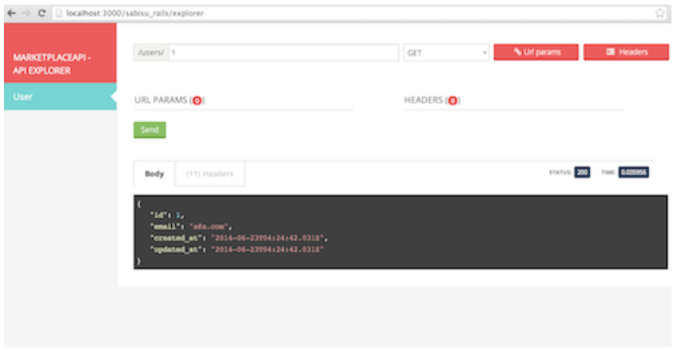
Trust me, I spent hours trying to solve this and wanted to point out this issue on purpose as it might save you some cups of coffee. The solution is fairly simple and is related to directory naming:

First move to the **`workspace`** directory or wherever the **`market_place_api`** directory is:

```
$ cd ~/workspace/
$ mv market_place_api marketplaceapi
```

Yes you can do this through the UI, but I feel more comfortable working through the terminal. Then we update the symbolic link we created on Section 1.3.1 to point to the new directory:

```
$ cd ~/.pow
$ ln -s ~/workspace/marketplaceapi
```



```
{
  "id": 1,
  "email": "a@a.com",
  "created_at": "2014-06-23T04:24:42.031Z",
  "updated_at": "2014-06-23T04:24:42.031Z"
}
```

Figure 3.2: http://localhost:3000/sabisu_rails/explorer

And just keep things organised we delete the old symbolic link:

```
$ rm market_place_api
```

Then we need to update the `base_api_uri` on the `sabisu_rails.rb` initialiser file as well:

```
config.base_api_uri = 'api.marketplaceapi.dev'
```

Now if we visit http://localhost:3000/sabisu_rails/explorer (Figure 3.2) and add an `id` number of the field, in this case 1, and hit `send` we should see the json response from the server.

If you have similar issues with `prax` on Linux, the solution above should also do the job.

Remember you can always review the code base on github or shout me a tweet

3.4 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 Chapter 4 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!

Feeling like want to share your achievement, I'll be glad to read about it:

I just finished modeling the users for my api of Api on Rails tutorial by @kurenn!

Chapter 4

Refactoring tests

In Chapter 3 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 https://github.com/kurenn/market_place_api.git -b chapter3
```

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

So let' take a look to the `users_controller_spec.rb` file (Listing 4.1):

Listing 4.1: Users controller spec (*app/controllers/api/v1/users_controller.rb*)

```
require 'spec_helper'

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

  describe "GET #show" do
    before(:each) do
      @user = FactoryGirl.create :user
      get :show, 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 { should respond_with 200 }
  end

  describe "POST #create" do

    context "when is successfully created" do
      before(:each) do
        @user_attributes = FactoryGirl.attributes_for :user
        post :create, { 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 { should respond_with 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, { 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 whye 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 { should respond_with 422 }
    end
  end

  describe "PUT/PATCH #update" do
    before(:each) do
      @user = FactoryGirl.create :user
    end
    context "when is successfully updated" do
      before(:each) do
        patch :update, { 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 { should respond_with 200 }
    end

    context "when is not created" do
      before(:each) do
        patch :update, { 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 whye 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 { should respond_with 422 }
    end
  end

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

    it { should respond_with 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(Section 4.1).

- The `format` param is sent on every request.(Section 4.2). Although is not a bad practice perse, it is better if you handle the response type through headers.

So let's add a method for handling the `json` response, but before we continue, and 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
```

4.1 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 (Listing 4.2):

Listing 4.2: Request helpers JSON parse (*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 organised. The next step here is to update the `users_controller_spec.rb` file to use the method. A quick example is presented below:

```
it "returns the information about a reporter on a hash" do
  user_response = json_response # this is the updated line
  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, 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:

```
Dir[Rails.root.join("spec/support/**/*.rb")].each { |f| require f }
```

So what is missing?, the problem in here is we have to include this methods into rspec as controller type helpers. This is done by adding a single line of code on the `spec_helper.rb` file within the `Rspec.configure` block (Listing 4.3):

Listing 4.3: Include json helpers into Rspec (*spec/spec_helper.rb*)

```
RSpec.configure do |config|
  .
  .
  .
  #Including to test requests
  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"
[chapter4 26b7e51] Refactors the json parse method
 3 files changed, 17 insertions(+), 7 deletions(-)
 create mode 100644 spec/support/request_helpers.rb
```

4.2 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:

```
describe Api::V1::UsersController do
  # we concatenate the json format
  before(:each) { request.headers['Accept'] = "application/vnd.marketplace.v1, #{Mime::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:

```
describe Api::V1::UsersController do
  before(:each) { request.headers['Accept'] = "application/vnd.marketplace.v1, #{Mime::JSON}" }
  # now we added this line
  before(:each) { request.headers['Content-Type'] = Mime::JSON.to_s }
  .
  .
  .
```

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 0.19013 seconds
13 examples, 0 failures
```

As always, this is a good time to commit:

4.3 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 3 custom headers being added before each request:

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

This is good, but not good enough, because we will have to add this 5 lines of code for each file, and if for some reason we are changing let's say the response type to `xml`, well you do the math. But don't worry I provide 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(Listing 4.4)

Listing 4.4: Include headers helpers into Rspec (*spec/support/request_helpers.rb*)

```
module Request
  module JsonHelpers
    def json_response
      @json_response ||= JSON.parse(response.body, symbolize_names: true)
    end
  end
  # Our headers helpers module
  module HeadersHelpers
    def api_header(version = 1)
      request.headers['Accept'] = "application/vnd.marketplace.v#{version}"
    end

    def api_response_format(format = Mime::JSON)
      request.headers['Accept'] = "#{request.headers['Accept']},#{format}"
      request.headers['Content-Type'] = format.to_s
    end

    def include_default_accept_headers
      api_header
      api_response_format
    end
  end
end
```

As you can see I broke the calls into 2 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 on the *Rspec.configure* block at spec_helper.rb 4.5 file, and make sure we specify the type to `:controller`, as we don't to run this on unit tests.

```
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_sp`
file and check that our tests are still passing.

You can review a full version of the `spec_helper.rb` file below (Listing 4.5):

Listing 4.5: Rspec helper (*spec/spec_helper.rb*)

```ruby
# This file is copied to spec/ when you run 'rails generate rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'rspec/autorun'

# Requires supporting ruby files with custom matchers and macros, etc,
# in spec/support/ and its subdirectories.
Dir[Rails.root.join("spec/support/**/*.rb")].each { |f| require f }

# Checks for pending migrations before tests are run.
# If you are not using ActiveRecord, you can remove this line.
ActiveRecord::Migration.check_pending! if defined?(ActiveRecord::Migration)

RSpec.configure do |config|
  # ## Mock Framework
  #
  # If you prefer to use mocha, flexmock or RR, uncomment the appropriate line:
  #
  # config.mock_with :mocha
  # config.mock_with :flexmock
  # config.mock_with :rr

  # Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
  config.fixture_path = "#{::Rails.root}/spec/fixtures"

  # If you're not using ActiveRecord, or you'd prefer not to run each of your
  # examples within a transaction, remove the following line or assign false
  # instead of true.
  config.use_transactional_fixtures = true

  # If true, the base class of anonymous controllers will be inferred
  # automatically. This will be the default behavior in future versions of
  # rspec-rails.
  config.infer_base_class_for_anonymous_controllers = false

  # Run specs in random order to surface order dependencies. If you find an
  # order dependency and want to debug it, you can fix the order by providing
  # the seed, which is printed after each run.
  #     --seed 1234
  config.order = "random"

  #Including to test requests
  config.include Request::JsonHelpers, :type => :controller
  config.include Request::HeadersHelpers, :type => :controller

  config.before(:each, type: :controller) do
    include_default_accept_headers
  end
```

```
end
```

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

```
$ git add .
$ git commit -am "Refactors test headers for each request"
[chapter4 ae10fe9] Refactors test headers for each request
 3 files changed, 21 insertions(+), 5 deletions(-)
```

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

4.4 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 Chapter 5 we will add the authentication mechanism we'll be using across the application as well as limiting the access for certain actions.

I'll be really excited to hear about you:

I just finished refactoring the test suite for my api of Api on Rails tutorial by @kurenn!

Chapter 5

Authenticating users

It's been a long way since you started, I hope you are enjoying this trip as much as me. On Chapter 4 we refactor our test suite and since we did not add much code, it didn't take to much. 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 https://github.com/kurenn/market_place_api.git -b chapter4
```

In this chapter things will get very interesting, and that is because we are going to set up our authentication mechanism. In my opinion this is going to be one of the most interesting chapters, as 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
```

5.1 Stateless and sign in failure

Before we go any further, something must be clear, an API does not handle sessions and 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.

5.1.1 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_authentication_token_to_users auth_token:string
```

As a good practice I like to setup `string` values to an empty string and in this case let's add an index with a unique condition to true. This way we warranty there are no users with the same token, at a database level, so let's do that:

```
class AddAuthenticationTokenToUsers < ActiveRecord::Migration
  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 & rake db:test:prepare
==  AddAuthenticationTokenToUsers: migrating =================================
-- add_column(:users, :auth_token, :string, {:default=>""})
   -> 0.0050s
-- add_index(:users, :auth_token, {:unique=>true})
   -> 0.0013s
==  AddAuthenticationTokenToUsers: migrated (0.0065s) ========================
```

Now it would be a good time to add some response and uniqueness tests to our `user` model spec (Listing 5.1)

Listing 5.1: Response and uniqueness tests (*spec/models/user_spec.rb*)

```
  # we test the user actually respond to this attribute
  it { should respond_to(:auth_token) }
  .
  .
  .
  # we test the auth_token is unique
  it { should validate_uniqueness_of(:auth_token)}
```

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

```
class User < ActiveRecord::Base
  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 (Listing 5.2).

Listing 5.2: Generate token spec (*spec/models/user_spec.rb*)

```
.
.
.
  it { should validate_uniqueness_of(:auth_token)}

  describe "#generate_authentication_token!" do
    it "generates a unique token" do
      Devise.stub(:friendly_token).and_return("auniquetoken123")
      @user.generate_authentication_token!
      expect(@user.auth_token).to eql "auniquetoken123"
    end

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

The tests initially fail, as expected:

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

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:

```
  def generate_authentication_token!
    begin
      self.auth_token = Devise.friendly_token
    end while self.class.exists?(auth_token: auth_token)
  end
```

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

```
  before_create :generate_authentication_token!
```

After doing this we should have all of our tests passing:

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

Finished in 0.13354 seconds
12 examples, 0 failures
```

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

```
$ git add db/migrate/20140629060140_add_authentication_token_to_users.rb
$ git add app/models/user.rb
$ git add db/schema.rb
$ git add spec/models/user_spec.rb
$ git commit -m "Adds user authentication token"
[chapter5 076e03e] Adds user authentication token
 4 files changed, 35 insertions(+), 1 deletion(-)
 create mode 100644 db/migrate/20140629060140_add_authentication_token_to_users.rb
```

5.1.2 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(Section 5.1.2) and the sign out will be handled by a *DELETE* request to the `destroy` action(Recall for the Listing 2.2 about HTTP verbs).

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 Listing 5.3 and Listing 5.4.

Listing 5.3: Sessions controller name update (*app/controllers/api/v1/sessions_controller.rb*)

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

end
```

Listing 5.4: Sessions controller spec name update (*spec/controllers/api/v1/sessions_controller_spec.rb*)

```
require 'spec_helper'

describe Api::V1::SessionsController do

end
```

Sign in success

Our first stop will be the `create` action, but first and as usual let's generate our tests (Listing 5.5)

Listing 5.5: Create tests for the ‘create‘ action (*spec/controllers/api/v1/sessions_controller_spec.rb*)

```
require 'spec_helper'

describe Api::V1::SessionsController do

  describe "POST #create" do

   before(:each) do
    @user = FactoryGirl.create :user
   end

    context "when the credentials are correct" do

      before(:each) do
        credentials = { email: @user.email, password: "12345678" }
        post :create, { session: credentials }
      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 { should respond_with 200 }
    end

    context "when the credentials are incorrect" do

      before(:each) do
        credentials = { email: @user.email, password: "invalidpassword" }
        post :create, { session: credentials }
      end

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

      it { should respond_with 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 (Listing 5.6). But before that we will add the end points to our `route.rb` file (both the `create` and `destroy` end point).

```
.
.
.
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  resources :users, :only => [:show, :create, :update, :destroy]
  resources :sessions, :only => [:create, :destroy]
end
.
.
.
```

Listing 5.6: Sessions controller with ‘create‘ action (*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, store: false
      user.generate_authentication_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:

```
.
.
.
#Including to test requests
config.include Request::JsonHelpers, :type => :controller
config.include Request::HeadersHelpers, :type => :controller
config.include Devise::TestHelpers, :type => :controller
.
.
.
```

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

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

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, and 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.

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 (Listing 5.7)

Listing 5.7: Destroy action sessions spec (*spec/controllers/api/v1/sessions_controller_spec.rb*)

```
describe "DELETE #destroy" do

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

  it { should respond_with 204 }

end
```

As you can see the `test` is super simple, now we just need to implement the necessary code to make our tests pass (Listing 5.8)

Listing 5.8: Destroy action sessions (*app/controllers/api/v1/sessions_controller.rb*)

```
def destroy
  user = User.find_by(auth_token: params[:id])
  user.generate_authentication_token!
  user.save
  head 204
end
```

In this case we are expecting an `id` to be sent on the request, which has to correspond to the *user authentication token*, on Section 5.2, 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!, but in the meantime commit the changes:

```
$ git add .
$ git commit -m "Adds destroy session action added"
[chapter5 cb8b47a] Adds destroy session action added
 3 files changed, 21 insertions(+), 1 deletion(-)
```

5.2 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.(See documentation on this for more details).

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, and 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.(Listing 5.9)

Listing 5.9: Authenticable spec current user method (*spec/controllers/concerns/authenticable_spec.rb*)

```
require 'spec_helper'

class Authentication
  include Authenticable
end

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

  describe "#current_user" do
    before do
      @user = FactoryGirl.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
```

If you are wondering, why in the hell we created a 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, like the request shown above.

Our tests should fail:

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

Let's implement the necessary code (Listing 5.10):

Listing 5.10: Authenticable current user method (*app/controllers/concerns/authenticable.rb*)

```
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 passing:

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

Finished in 0.03838 seconds
1 example, 0 failures
```

Now we just need to include the `Authenticable` module into the `ApplicationContr`

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

  include Authenticable

end
```

This would be a good time to commit the changes:

```
$ git add .
$ git commit -m "Adds authenticable module for managing authentication methods"
[chapter5 1d049c1] Adds authenticable module for managing authentication methods
 3 files changed, 30 insertions(+)
 create mode 100644 app/controllers/concerns/authenticable.rb
 create mode 100644 spec/controllers/concerns/authenticable_spec.rb
```

5.3 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(Listing 5.11).

Listing 5.11: Authenticable authenticate with token method (*spec/controllers/concerns/authenticable_spec.rb*)

```
describe Authenticable do
  .
  .
  .
  describe "#authenticate_with_token" do
    before do
      @user = FactoryGirl.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 {  should respond_with 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 (Listing 5.12).

Listing 5.12: Authenticable authenticate with token method (*app/controllers/concerns/authenticable.rb*)

```
module Authenticable

  # Devise methods overwrites
  .
  .
  .
  def authenticate_with_token!
    render json: { errors: "Not authenticated" },
                  status: :unauthorized unless current_user.present?
  end

end
```

At this point, we just built a super simple authorization mechanism to prevent unsigned users to access the api. We just need to update the **`users_controller.rb`** file with the **`current_user`** method and prevent the access with the **`authenticate_with`**

Let's commit this changes and keep moving:

```
$ git commit -m "Adds the authenticate with token method to handle access to actions"
[chapter5 99e869d] Adds the authenticate with token method to handle access to actions
 2 files changed, 22 insertions(+)
```

5.4 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 (Listing 5.13). 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.

Listing 5.13: Implementation of current user for the update action (*app/controllers/users_controller.rb*)

```
  .
  .
  .
  def update
    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
  .
  .
  .
```

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

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

Failures:
.
.
.
```

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

```
describe "PUT/PATCH #update" do
  before(:each) do
    @user = FactoryGirl.create :user
    request.headers['Authorization'] =  @user.auth_token
  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 on Chapter 4. Listing 5.14

Listing 5.14: Authorization header method (*spec/support/request_helpers.rb*)

```
module HeadersHelpers
  .
  .
  .
  def api_authorization_header(token)
    request.headers['Authorization'] =  token
  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 Listing 5.15

Listing 5.15: Destroy action with current$_{u}ser$(*app/controllers/api/v1/users_controller.rb*)

```
.
.
.
def destroy
  current_user.destroy
  head 204
end
.
.
.
```

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

```
describe "DELETE #destroy" do
  before(:each) do
    @user = FactoryGirl.create :user
    api_authorization_header @user.auth_token #we added this line
    delete :destroy, id: @user.auth_token
  end

  it { should respond_with 204 }

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 2 actions.

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 Listing 5.16

Listing 5.16: Filter actions for users controller (*app/controllers/api/v1/users_controller.rb*)

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

Our tests should still be passing. And from now on everytime 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"
[chapter5 7c19c79] Adds authorization for the users controller
 3 files changed, 10 insertions(+), 4 deletions(-)
```

Lastly but not least we will finish the chapter by refactoring the `authenticate_with_t` method, it is really a small enhancement, but it will make the method more descriptive. You'll see what I mean in a minute(Listing 5.18), but first things first, let's add some specs.

Listing 5.17: User signed in method spec (*spec/controllers/concerns/authenticable_spec.rb*)

```
describe Authenticable do
  .
  .
  .
  describe "#user_signed_in?" do
    context "when there is a user on 'session'" do
      before do
        @user = FactoryGirl.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 = FactoryGirl.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, and as I mentioned early it is just for visual clarity. But let's keep going and add the implementation. (Listing 5.18)

Listing 5.18: User signed in method (*app/controllers/concerns/authenticable.rb*)

```
module Authenticable
  .
  .
  .
  def authenticate_with_token!
    render json: { errors: "Not authenticated" },
                status: :unauthorized unless user_signed_in?
  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, which 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:

```
$ bundle exec rspec/spec/controllers/concerns/authenticable.rb
.....

Finished in 0.08268 seconds
5 examples, 0 failures

Randomized with seed 59972
```

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"
```

5.5 Conclusion

Yei! 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 and 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.

In the mean time let me know how you doing and shout me a tweet:

I can authenticate users for my api thanks to Api on Rails tutorial by @kurenn!

Chapter 6

User products

On Chapter 5 we implemented the authentication mechanism 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 customise 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, but for those who doesn't, we will associated the **`User`** to the **`Product`** model using the **`has_many`** and **`belongs_to`** active record methods. Recall (Figure 2.1).

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 https://github.com/kurenn/market_place_api.git -b chapter5
```

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

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

6.1 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

6.1.1 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, and last but not least 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/20140718160927_create_products.rb
      create    app/models/product.rb
      invoke    rspec
      create      spec/models/product_spec.rb
      invoke      factory_girl
      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:

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :title, default: ""
      t.decimal :price, default: 0.0
      t.boolean :published, default: false
      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 in Listing 6.1:

Listing 6.1: Product model bare bones spec (*spec/models/product_spec.rb*)

```
require 'spec_helper'

describe Product do
  let(:product) { FactoryGirl.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) }

  it { should not_be_published }
end
```

Remember to migrate the database and prepare the test one so we get out tests green:

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

Make sure the tests pass:

```
$ bundle exec rspec spec/modes/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 (Listing 3.3), now it is time to do the same with the `product` model. See Listing 6.2

Listing 6.2: Product factory bare bones (*spec/factories/products.rb*)

```
FactoryGirl.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 on Section 6.1.3.

6.1.2 Product validations

As we saw with the user, validations are an important part when building any kind of application, this way we 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`. In Listing 6.3 you can see what I'm talking about.

Listing 6.3: Product validation specs (*spec/models/product_spec.rb*)

```
describe Product do
  let(:product) { FactoryGirl.build  :product }
  subject { product }
  .
  .
  .
  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:

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

end
```

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"
[chapter6 f774173] Adds product model bare bones along with some validations
 5 files changed, 58 insertions(+), 1 deletion(-)
 create mode 100644 app/models/product.rb
 create mode 100644 db/migrate/20140718160927_create_products.rb
 create mode 100644 spec/factories/products.rb
 create mode 100644 spec/models/product_spec.rb
```

6.1.3 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?:

```
FactoryGirl.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, as FactoryGirl 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. See Listing 6.4

Listing 6.4: Product association specs (*spec/models/product_spec.rb*)

```
describe Product do
  let(:product) { FactoryGirl.build :product }
  subject { product }
  .
  .
  .
  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:

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

  # we added this line
  belongs_to :user

end
```

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

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

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:

```
describe User do
  .
  .
  .
  it { should have_many(:products) }
  .
  .
  .

end
```

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

```
class User < ActiveRecord::Base
  .
  .
  .
  before_create :generate_authentication_token!

  has_many :products
  .
  .
  .
end
```

Now if we run the user specs, they should be all nice and green:

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

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 in Listing 6.5:

Listing 6.5: User products destroy dependency specs (*spec/models/user_spec.rb*)

```
describe User do
  before { @user = FactoryGirl.build(:user) }

  subject { @user }
  .
  .
  .
  describe "#products association" do

    before do
      @user.save
      3.times { FactoryGirl.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)).to raise_error ActiveRecord::RecordNotFound
      end
    end
  end
end
```

The necessary code to make the code on Listing 6.5 to pass is just an option on the `has_many` association method:

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :products, dependent: :destroy
  .
  .
  .
end
```

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

```
$ bundle exec rspec spec/
```

Let's commit this and move on to the next Section 6.2

```
$ git add .
$ git commit -m "Finishes modeling the product model along with user associations"
[chapter6 13795aa] Finishes modeling the product model along with user associations
 5 files changed, 26 insertions(+), 2 deletions(-)
```

6.2 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 customise 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. Listing 6.6

Listing 6.6: Products controller bare bones (*app/controllers/api/v1/products_controller.rb*)

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

And the spec file should look like the one on Listing 6.7

Listing 6.7: Products controller spec bare bones (*spec/controllers/api/v1/products_controller_spec.rb*)

```
require 'spec_helper'

describe Api::V1::ProductsController do

end
```

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

6.2.1 Show action for products

As usual we begin by adding some product `show` controller specs. Listing 6.8. 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, as recall on Listing 3.6.

Listing 6.8: Product show action specs (*spec/controllers/api/v1/products_controller_spec.rb*)

```
describe Api::V1::ProductsController do
  describe "GET #show" do
    before(:each) do
      @product = FactoryGirl.create :product
      get :show, 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 { should respond_with 200 }
  end

end
```

We then add the code to make the test pass:

```
class Api::V1::ProductsController < ApplicationController
  respond_to :json

  def show
    respond_with 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:

```
  namespace :api, defaults: { format: :json },
                  constraints: { subdomain: 'api' }, path: '/'  do
    scope module: :v1,
                  constraints: ApiConstraints.new(version: 1, default: true) do
      # We are going to list our resources here
      resources :users, :only => [:show, :create, :update, :destroy]
      resources :sessions, :only => [:create, :destroy]
      resources :products, :only => [:show]
    end
  end
```

Now we make sure the tests are nice and green:

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

Finished in 0.1068 seconds
2 examples, 0 failures

Randomized with seed 25484
```

As you may notice already the specs and implementation are very simple, actually they behave the same as the users, recall Listing 3.6

6.2.2 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. Listing 6.9:

Listing 6.9: Product index action specs (*spec/controllers/api/v1/products_controller_spec.rb*)

```
describe Api::V1::ProductsController do
  .
  .
  .
  describe "GET #index" do
    before(:each) do
      4.times { FactoryGirl.create :product }
      get :index
    end

    it "returns 4 records from the database" do
      products_response = json_response
      expect(products_response[:products]).to have(4).items
    end

    it { should respond_with 200 }
  end
end
```

Let's move into the implementation, which for now is going to be a sad `all` class method. Listin 6.10

Listing 6.10: Product index action (*spec/controllers/api/v1/products_controller.rb*)

```
class Api::V1::ProductsController < ApplicationController
  respond_to :json

  def index
    respond_with Product.all
  end
  .
  .
  .
end
```

And remember, you have to add the corresponding route:

```
resources :products, :only => [:show, :index]
```

In next chapters we will improve this endpoint, and give it the ability to receive parameters to filter them.

6.2.3 Exploring with Sabisu

Hey, remember back in Chapter 3 on Section 3.3 we integrate a gem called `sabisu_rails`, well let's customize it to map the products endpoints. This is easily done by going to the `sabisu_rails.rb` initialiser file and modify line 25 like the one below:

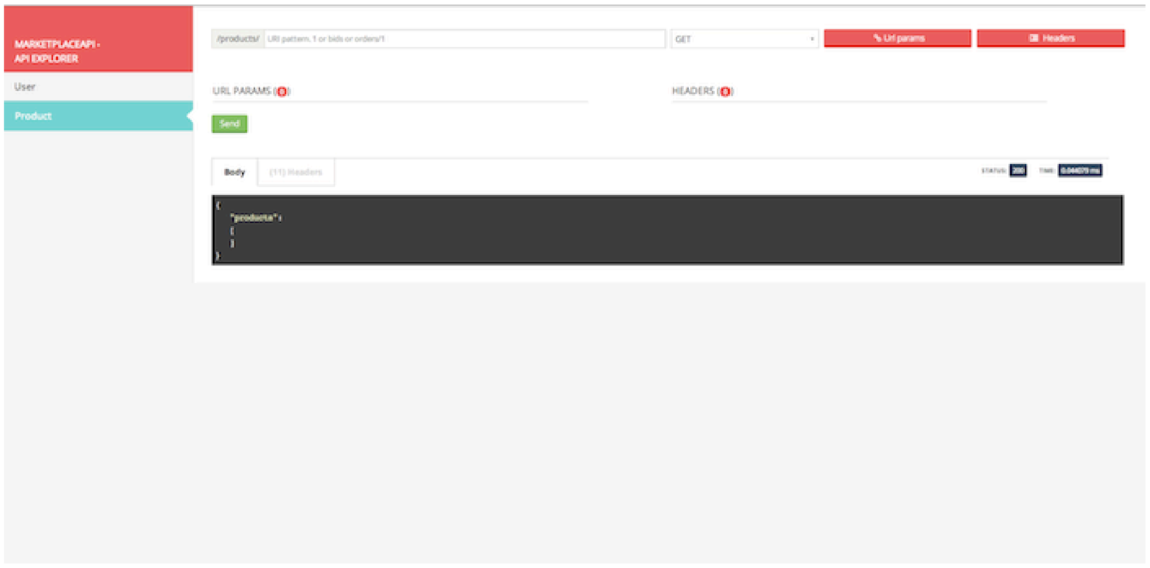
```
SabisuRails.setup do |config|
  .
  .
  .
   config.resources = [:users, :products]
    .
    .
    .
 end
```

If you run the rails server from the command line, and visit http://localhost:3000/sabisu_r the product resource should be selected and the screen should look like Figure 6.1

You can create a bunch of products through the console using FactoryGirl and see a reasonable response.

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 -am "Adds public product endpoints(show, index)"
[chapter6 185adcb] Adds public product endpoints(show, index)
```



```
{
  "products":
  [
  ]
}
```

Figure 6.1: http://localhost:3000/sabisu_rails/explorer?explorer%5Bresource%5D=products

```
4 files changed, 46 insertions(+), 1 deletion(-)
create mode 100644 app/controllers/api/v1/products_controller.rb
create mode 100644 spec/controllers/api/v1/products_controller_spec.rb
```

6.2.4 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. Listing 6.11

Listing 6.11: Products create action spec (*spec/controllers/api/v1/products_controller_spec.rb*)

```
describe Api::V1::ProductsController do
  .
  .
  .
  describe "POST #create" do
    context "when is successfully created" do
      before(:each) do
        user = FactoryGirl.create :user
        @product_attributes = FactoryGirl.attributes_for :product
        api_authorization_header user.auth_token
        post :create, { 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 { should respond_with 201 }
    end

    context "when is not created" do
      before(:each) do
        user = FactoryGirl.create :user
        @invalid_product_attributes = { title: "Smart TV", price: "Twelve dollars" }
        api_authorization_header user.auth_token
        post :create, { 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 whye the user could not be created" do
        product_response = json_response
        expect(product_response[:errors][:price]).to include "is not a number"
      end

      it { should respond_with 422 }
    end
  end
end
```

Wow!, we added a bunch of code, but if you recall from Section 3.2.2, 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, { 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. Listing 6.12

Listing 6.12: Products create action (*spec/controllers/api/v1/products_controller.rb*)

```
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_toke` 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?, because 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:

```
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  resources :users, :only => [:show, :create, :update, :destroy] do
    # this is the line
    resources :products, :only => [:create]
  end
  resources :sessions, :only => [:create, :destroy]
  resources :products, :only => [:show, :index]
end
```

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

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

Finished in 0.26459 seconds
9 examples, 0 failures
Randomized with seed 48949
```

6.2.5 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:

```
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  resources :users, :only => [:show, :create, :update, :destroy] do
    # this is the line
    resources :products, :only => [:create, :update]
  end
  resources :sessions, :only => [:create, :destroy]
  resources :products, :only => [:show, :index]
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, actually belongs to the user, so we will fetch that product from the **`user.products`** association provided by rails.

First we add some specs, Listing 6.13

Listing 6.13: Products update action spec (*spec/controllers/api/v1/products_controller_spec.rb*)

```
  describe "PUT/PATCH #update" do
    before(:each) do
      @user = FactoryGirl.create :user
      @product = FactoryGirl.create :product, user: @user
      api_authorization_header @user.auth_token
    end

    context "when is successfully updated" do
      before(:each) do
        patch :update, { 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 { should respond_with 200 }
    end

    context "when is not updated" do
      before(:each) do
        patch :update, { 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 whye the user could not be created" do
        product_response = json_response
        expect(product_response[:errors][:price]).to include "is not a number"
      end

      it { should respond_with 422 }
    end
  end
```

The tests may look complex, but take a second peek, they are almost the same as the users on Listing 3.11, the only difference here is the nested routes as we saw on Section 6.2.4, 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 Listing 6.14:

Listing 6.14: Products update action (*spec/controllers/api/v1/products_controller.rb*)

```
class Api::V1::ProductsController < ApplicationController
  before_action :authenticate_with_token!, only: [: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 unauthorised user to update a product.

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

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

Finished in 0.36063 seconds
14 examples, 0 failures

Randomized with seed 24040
```

6.2.6 Destroying products

Our last stop for the products endpoints, will be the **`destroy`** action and 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 fecth 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:

```
scope module: :v1, constraints: ApiConstraints.new(version: 1, default: true) do
  # We are going to list our resources here
  resources :users, :only => [:show, :create, :update, :destroy] do
    resources :products, :only => [:create, :update, :destroy]
  end
  resources :sessions, :only => [:create, :destroy]
  resources :products, :only => [:show, :index]
end
```

After this, we have to add some tests as shown on Listing 6.15:

Listing 6.15: Products destroy action spec (*spec/controllers/api/v1/products_controller_spec.rb*)

```
  describe "DELETE #destroy" do
    before(:each) do
      @user = FactoryGirl.create :user
      @product = FactoryGirl.create :product, user: @user
      api_authorization_header @user.auth_token
      delete :destroy, { user_id: @user.id, id: @product.id }
    end

    it { should respond_with 204 }
  end
```

Now we simply add the necessary code to make the tests pass, see Listing 6.16:

Listing 6.16: Products destroy action (*spec/controllers/api/v1/products_controller.rb*)

```
def destroy
  product = current_user.products.find(params[:id])
  product.destroy
  head 204
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. Listing 6.14

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

Finished in 0.44976 seconds
15 examples, 0 failures

Randomized with seed 33241
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Adds the products create, update and destroy action nested on the user resourc
[chapter6 73057d2] Adds the products create, update and destroy action nested on the user resou
 3 files changed, 124 insertions(+), 1 deletion(-)
```

6.3 Populating the database

Before we continue with more code, let's populate the database with some fake data. Thanfully 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 `FactoryGirl` gem:

```
Loading development environment (Rails 4.0.2)
2.1.0 :001 > 20.times { FactoryGirl.create :product }
```

Oops, you probably have some errors showing up:

```
NameError: uninitialized constant FactoryGirl
  from (irb):1:in `block in irb_binding'
  from (irb):1:in `times'
  from (irb):1
  .
  .
  .
  from bin/rails:4:in `require'
  from bin/rails:4:in `<main>'
```

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:

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

group :test do
```

```
  gem "rspec-rails"
  gem "factory_girl_rails"
  gem 'ffaker'
  gem "shoulda-matchers"
end
.
.
.
```

You see where the problem is?. If you pay attention you will notice that the `factory_girl_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:

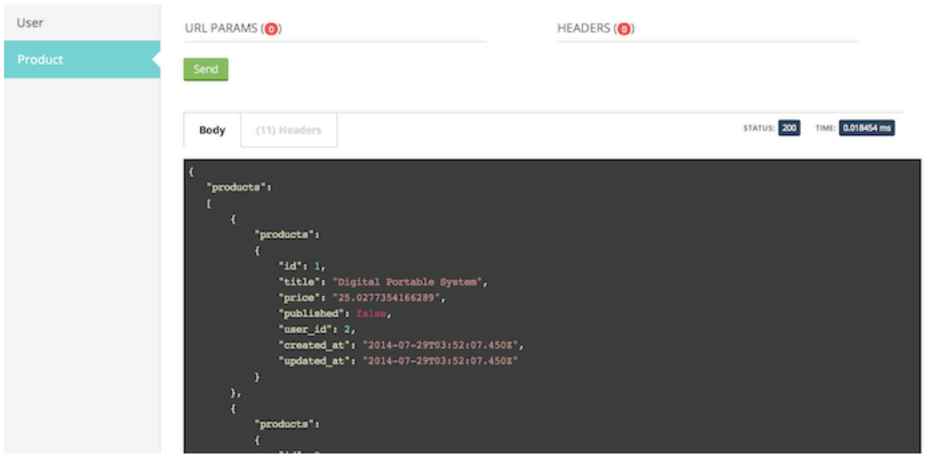
```
group :development do
  gem 'sqlite3'
end

group :development, :test do
  gem "factory_girl_rails"
  gem 'ffaker'
end

group :test do
  gem "rspec-rails"
  gem "shoulda-matchers"
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 4.0.2)
2.1.0 :001 > 20.times { FactoryGirl.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:



```
{
  "products":
  [
    {
      "products":
      {
        "id": 1,
        "title": "Digital Portable System",
        "price": "25.0277354166289",
        "published": false,
        "user_id": 2,
        "created_at": "2014-07-29T03:52:07.450Z",
        "updated_at": "2014-07-29T03:52:07.450Z"
      }
    },
    {
      "products":
      {
```

Figure 6.2: http://localhost:3000/sabisu_rails/explorer?explorer%5Bresource%5D=products

```
$ git add .
$ git commit -m "Updates test environment factory gems to work on development"
[chapter6 6da3f8f] Updates test environment factory gems to work on development
 1 file changed, 5 insertions(+), 2 deletions(-)
```

If you check the output now for the products using `sabisu`, it should look like the Figure 6.2. It does not look nice isn't it?, we have to make some customisation to the output using `active_model_serializers`.

6.4 Conclusion

On the next chapter we, will focus on customising the output from the `user` and `product` models using the active model serializers gem which will help us to easily filter the attributes to display, or handle associations as embebed 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. In the mean time, tell me how are you doing, I'll be glad to hear from you:

I just finished modeling the products for my api of Api on Rails tutorial by @kurenn!

Chapter 7

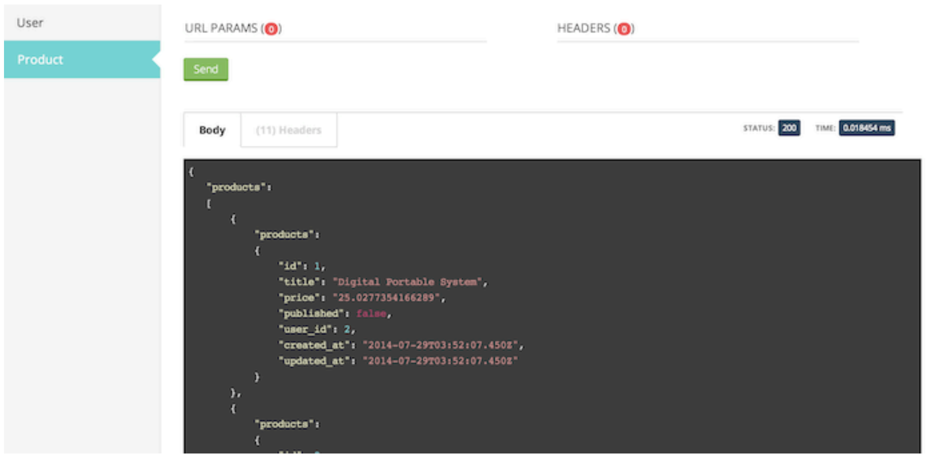
JSON with Active Model Serializers

In Chapter 6 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 like the Figure 7.1, which doesn't look nice, as the json output should render just an array of products, with the `products` as the root key, something like Listing 7.1

Listing 7.1: Desired products json output

```
"products": [
  {
    "id": 1,
    "title": "Digital Portable System",

  },
  {
    "id": 2,
    "title": "Plasma TV",
  }

]
```

This is further explained in the JSON API website.



```
{
  "products":
  [
      {
          "products":
          {
              "id": 1,
              "title": "Digital Portable System",
              "price": "25.0277354166289",
              "published": false,
              "user_id": 2,
              "created_at": "2014-07-29T03:52:07.450Z",
              "updated_at": "2014-07-29T03:52:07.450Z"
          }
      },
      {
          "products":
          {
```

Figure 7.1: *http://localhost:3000/sabisu_rails/explorer?explorer%5Bresource%5D=products*

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 customise the json output using the `active_model_serialize` 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/kurenn/market_place_api.git -b chapter6
```

Let's branch out this chapter with:

```
$ git checkout -b chapter7
Switched to a new branch 'chapter7'
```

7.1 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`:

```
.
.
.
gem 'active_model_serializers', git: 'git@github.com:rails-api/active_model_serializers.git', b
.
.
.
```

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

7.2 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 the one on Listing 7.2:

Listing 7.2: User model serializer bare bones (*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:

```
$ bundle exec rspec spec/controllers/api/v1/users_controller_spec.rb
....F.....F.F
.
.
.
Finished in 0.2492 seconds
13 examples, 3 failures
```

If you take a quick look at the **`users_controller`**, you may have something like Listing 3.7 for the show action, and as it is mention on the gem documentation, Box 7.1

Box 7.1. Active model serializers gem

> *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 serialiser 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 serialise into the **`user_serializer.rb`** and update the **`users_controller_spec.rb`** file:

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

```ruby
require 'spec_helper'

describe Api::V1::UsersController do
  describe "GET #show" do
    before(:each) do
      @user = FactoryGirl.create :user
      get :show, id: @user.id
    end

    it "returns the information about a reporter on a hash" do
      user_response = json_response[:user]
      expect(user_response[:email]).to eql @user.email
    end

    it { should respond_with 200 }
  end

  describe "POST #create" do

    context "when is successfully created" do
      before(:each) do
        @user_attributes = FactoryGirl.attributes_for :user
        post :create, { user: @user_attributes }
      end

      it "renders the json representation for the user record just created" do
        user_response = json_response[:user]
        expect(user_response[:email]).to eql @user_attributes[:email]
      end

      it { should respond_with 201 }
    end

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

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

      it "renders the json errors on whye the user could not be created" do
        user_response = json_response
        expect(user_response[:errors][:email]).to include "can't be blank"
      end

      it { should respond_with 422 }
    end
  end

  describe "PUT/PATCH #update" do
    before(:each) do
      @user = FactoryGirl.create :user
      api_authorization_header @user.auth_token
    end

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

      it "renders the json representation for the updated user" do
        user_response = json_response[:user]
        expect(user_response[:email]).to eql "newmail@example.com"
      end

      it { should respond_with 200 }
    end

    context "when is not updated" do
      before(:each) do
        patch :update, { id: @user.id, user: { email: "bademail.com" } }
      end

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

      it "renders the json errors on whye the user could not be created" do
        user_response = json_response
        expect(user_response[:errors][:email]).to include "is invalid"
      end

      it { should respond_with 422 }
    end
  end

  describe "DELETE #destroy" do
    before(:each) do
```

```
      @user = FactoryGirl.create :user
      api_authorization_header @user.auth_token
      delete :destroy, { id: @user.id }
    end

    it { should respond_with 204 }

  end
end
```

If you pay enough attention and as I mentioned before the gem adds a root for the json object that corresponds to the serialised object name, in this case `user`. You can check it out with sabisu.

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

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

Finished in 0.25168 seconds
13 examples, 0 failures

Randomized with seed 10177
```

Let's commit the changes:

```
$ git add .
$ git commit -am "Adds user serializer for customizing the json output"
[chapter7 827b1f9] Adds user serializer for customizing the json output
 2 files changed, 6 insertions(+), 3 deletions(-)
 create mode 100644 app/serializers/user_serializer.rb
```

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

7.3 Serialise 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 Section 7.2:

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

If we run the tests now, we should have 3 specs failing:

```
Failures:

  1) Api::V1::ProductsController GET #show returns the information about a reporter on a hash
     Failure/Error: expect(product_response[:title]).to eql @product.title

       expected: "Direct Digital Compressor"
            got: nil

       (compared using eql?)
     # ./spec/controllers/api/v1/products_controller_spec.rb:13:in `block (3 levels) in <top (r

  2) Api::V1::ProductsController PUT/PATCH #update when is successfully updated renders the jso
     Failure/Error: expect(product_response[:title]).to eql "An expensive TV"

       expected: "An expensive TV"
            got: nil

       (compared using eql?)
     # ./spec/controllers/api/v1/products_controller_spec.rb:86:in `block (4 levels) in <top (r

  3) Api::V1::ProductsController POST #create when is successfully created renders the json rep
     Failure/Error: expect(product_response[:title]).to eql @product_attributes[:title]

       expected: "Performance Mount"
            got: nil

       (compared using eql?)
     # ./spec/controllers/api/v1/products_controller_spec.rb:44:in `block (4 levels) in <top (r

Finished in 0.34357 seconds
15 examples, 3 failures
```

Let's go one spec at a time. First we go to line 13 on the `products_controller_spec` an update the `product_response` variable:

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

Then we jump to line 86 and make the same change as the last one:

```
.
.
.
it "renders the json representation for the updated user" do
  product_response = json_response[:product]
  expect(product_response[:title]).to eql "An expensive TV"
end
.
.
.
```

Lastly we jump back to line 44, and yeah we make the same little change we've been doing so far:

```
.
.
.
it "renders the json representation for the product record just created" do
  product_response = json_response[:product]
  expect(product_response[:title]).to eql @product_attributes[:title]
end
.
.
.
```

Now if we run the tests we should be back to green:

```
...............

Finished in 0.3121 seconds
15 examples, 0 failures

Randomized with seed 37796
```

If you feel anxious on how the json outputs are right now, remember you can always use sabisu and play around with it.

Let's commit the changes and move on onto Section 7.4:

```
$ git add .
$ git commit -a "Adds product serializer for custom json output"
[chapter7 f8834ec] Adds product serializer for custom json output
 4 files changed, 8 insertions(+), 5 deletions(-)
 create mode 100644 app/serializers/product_serializer.rb
```

7.4 Sessions

If you run all of your test suite, you may notice that we have a failing test on the `sessions_controlles_spec.rb` file:

```
Failures:

  1) Api::V1::SessionsController POST #create when the credentials are correct returns the user
     Failure/Error: expect(json_response[:auth_token]).to eql @user.auth_token

       expected: "z-zkCoks3xycLAUyBXzf"
            got: nil

       (compared using eql?)
     # ./spec/controllers/api/v1/sessions_controller_spec.rb:20:in `block (4 levels) in <top (r

Finished in 0.82319 seconds
62 examples, 1 failure
```

This is because we change the output format for the json response, and as we have been doing it so far we have to update the spec as shown on Listing 7.3.

Listing 7.3: Sessions controller fix (*spec/controllers/api/v1/sessions_controllller$_spec.rb$*)

```
  .
  .
  .
  # line 18
      it "returns the user record corresponding to the given credentials" do
        @user.reload
        expect(json_response[:user][:auth_token]).to eql @user.auth_token
      end
  .
  .
  .
```

Now that we have everything nice and green let's commit the changes:

```
$ git add .
$ git commit -am "Fixes the sessions controller spec"
[chapter7 6342601] Fixes the sessions controller spec
 2 files changed, 2 insertions(+), 2 deletions(-)
```

7.5 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, and I'll to 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 Listing 7.4 and Listing 7.5

Listing 7.4: User model association (*app/models/user.rb*)

```
class User < ActiveRecord::Base
  .
  .
  .
  has_many :products, dependent: :destroy
  .
  .
  .
end
```

Listing 7.5: Product model association (*app/models/product.rb*)

```
class Product < ActiveRecord::Base
  .
  .
  .
  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 on Box 7.2.

Box 7.2. Association embeded object use case

Image an scenario where you are fetching the products from the api, but in this case 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, because 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 and 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 according to Box 7.2 we'll be embeding the user object into the product, let's start by adding some tests. We will just modify the `show` and `index` endpoints spec. Listing 7.6

Listing 7.6: Product with user embeded (*spec/controllers/api/v1/products_controller_spec.rb*)

```
describe Api::V1::ProductsController do

  describe "GET #show" do
    before(:each) do
      @product = FactoryGirl.create :product
      get :show, id: @product.id
    end
    .
    .
    .
    it "has the user as a embeded object" do
      product_response = json_response[:product]
      expect(product_response[:user][:email]).to eql @product.user.email
    end
    .
    .
    .
  end

  describe "GET #index" do
    before(:each) do
      4.times { FactoryGirl.create :product }
      get :index
    end
    .
    .
    .
    it "returns the user object into each product" do
      products_response = json_response[:products]
      products_response.each do |product_response|
        expect(product_response[:user]).to be_present
      end
    end
    .
    .
    .
  end
```

The implementation is really easy, we just need to add one line to the product serializer:

```
class ProductSerializer < ActiveModel::Serializer
  attributes :id, :title, :price, :published
  has_one :user #this is the line
end
```

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

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

Finished in 0.40742 seconds
17 examples, 0 failures
Randomized with seed 11585
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Embeds the user into the product json output"
[chapter7 e99ddf4] Embeds the user into the product json output
 2 files changed, 13 insertions(+)
```

7.5.1 Embeding 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 which in short loops the program until it runs out of memory and throws you and error or never respond you at all.

But don't worry not all is lost, we can easily solve this problem, and this is by embeding 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:

Listing 7.7: User product ids embeded spec(*spec/controllers/api/v1/users_controller_spec.rb*)

```
describe Api::V1::UsersController do
  describe "GET #show" do
```

```
    before(:each) do
      @user = FactoryGirl.create :user
      get :show, id: @user.id
    end
    .
    .
    .
    it "has the product ids as an embeded object" do
      user_response = json_response[:user]
      expect(user_response[:product_ids]).to eql []
    end
end
```

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

Listing 7.8: User product ids embeded (*app/serializers/user_serializer.rb*)

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

  has_many :products
end
```

We should have our tests passing:

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

Finished in 0.24388 seconds
14 examples, 0 failures

Randomized with seed 58058
```

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, see Listing 7.9

Listing 7.9: Products controller index with product$_idsspec($*spec/controllers/api/v1/products_controller_spec.rb*$)$

```
describe Api::V1::ProductsController do
  .
  .
  .
  describe "GET #index" do
    before(:each) do
      4.times { FactoryGirl.create :product }
    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
        products_response = json_response
        expect(products_response[:products]).to have(4).items
      end

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

      it { should respond_with 200 }
    end

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

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

As you can see from Listing 7.9 we just wrapped the index action into two separate contexts, one which will recevive the `product_ids`, and the old one we had which does not. Let's add the necessary code to make the tests pass:

```
class Api::V1::ProductsController < ApplicationController
  .
  .
  .
  def index
    products = params[:product_ids].present? ? Product.find(params[:product_ids]) : Product.all
    respond_with 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:

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

Finished in 0.49507 seconds
18 examples, 0 failures

Randomized with seed 31154
```

Let's commit the changes:

```
$ git commit -m "Embeds the products_ids into the user serialiser and fetches the correct produ
[chapter7 57d052b] Embeds the products_ids into the user serialiser and fetches the correct pro
 4 files changed, 41 insertions(+), 10 deletions(-)
```

7.6 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, but 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.

7.6.1 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:

Listing 7.10: Product model filter by title specs (*spec/models/product.rb*)

```
describe Product do
  .
  .
  .
  describe ".filter_by_title" do
    before(:each) do
      @product1 = FactoryGirl.create :product, title: "A plasma TV"
      @product2 = FactoryGirl.create :product, title: "Fastest Laptop"
      @product3 = FactoryGirl.create :product, title: "CD player"
      @product4 = FactoryGirl.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 apropiate comparison, in this case we'll use the lower case approach. Let's implement the necessary code:

Listing 7.11: Product model filter by title (*app/models/product.rb*)

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

end
```

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

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

Finished in 0.24312 seconds
12 examples, 0 failures

Randomized with seed 40477
```

7.6.2 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:

Listing 7.12: Product model above or equal to price specs (*spec/models/product.rb*)

```
describe Product do
  .
  .
  .
  describe ".above_or_equal_to_price" do
    before(:each) do
      @product1 = FactoryGirl.create :product, price: 100
      @product2 = FactoryGirl.create :product, price: 50
      @product3 = FactoryGirl.create :product, price: 150
      @product4 = FactoryGirl.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:

Listing 7.13: Product model above or equal to price (*app/models/product.rb*)

```
class Product < ActiveRecord::Base
  .
  .
  .
  scope :above_or_equal_to_price, lambda { |price|
    where("price >= ?", price)
  }

end
```

That should be sufficient, let's just verify everything is ok:

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

Finished in 0.29903 seconds
13 examples, 0 failures

Randomized with seed 59453
```

You can now imagine how the opposite method will behave, let's add the specs:

Listing 7.14: Product model below or equal to price specs (*spec/models/product.rb*)

```
describe Product do
  .
  .
  .
  describe ".below_or_equal_to_price" do
    before(:each) do
      @product1 = FactoryGirl.create :product, price: 100
      @product2 = FactoryGirl.create :product, price: 50
      @product3 = FactoryGirl.create :product, price: 150
      @product4 = FactoryGirl.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:

Listing 7.15: Product model below or equal to price (*app/models/product.rb*)

```
class Product < ActiveRecord::Base
  .
  .
  .
  scope :below_or_equal_to_price, lambda { |price|
    where("price <= ?", price)
  }

end
```

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

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

Finished in 0.31304 seconds
14 examples, 0 failures

Randomized with seed 59836
```

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 propietary of the product decides to update some of the data, the client always fetches the most updated records.

7.6.3 Sort by creation

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

Listing 7.16: Product model recent scope spec(*spec/models/product.rb*)

```
describe Product do
  .
  .
  .
  describe ".recent" do
    before(:each) do
      @product1 = FactoryGirl.create :product, price: 100
      @product2 = FactoryGirl.create :product, price: 50
      @product3 = FactoryGirl.create :product, price: 150
      @product4 = FactoryGirl.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:

Listing 7.17: Product model recent scope (*app/models/product.rb*)

```ruby
class Product < ActiveRecord::Base
  .
  .
  .
  scope :recent, -> {
    order(:updated_at)
  }

end
```

All of our tests should be green:

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

Finished in 0.32053 seconds
15 examples, 0 failures

Randomized with seed 36491
```

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"
[chapter7 3e9495b] Adds search scopes on the product model
 2 files changed, 78 insertions(+)
```

7.6.4 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:

Listing 7.18: Product model search method spec(*spec/models/product.rb*)

```ruby
describe Product do
  .
  .
  .
  describe ".search" do
    before(:each) do
      @product1 = FactoryGirl.create :product, price: 100, title: "Plasma tv"
      @product2 = FactoryGirl.create :product, price: 50, title: "Videogame console"
      @product3 = FactoryGirl.create :product, price: 150, title: "MP3"
      @product4 = FactoryGirl.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.

Listing 7.19: Product model search method (*app/models/product.rb*)

```ruby
class Product < ActiveRecord::Base
  .
  .
  .
  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:

```ruby
class Api::V1::ProductsController < ApplicationController
  .
  .
  .
  def index
    respond_with Product.search(params)
  end
  .
  .
  .
end
```

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

```
$ bundle exec rspec
...........................................................................
Finished in 1.31 seconds
75 examples, 0 failures
```

```
Randomized with seed 58705
```

Let's commit this:

```
$ git commit -am "Adds search class method to filter products"
[chapter7 2072f23] Adds search class method to filter products
 4 files changed, 49 insertions(+), 3 deletions(-)
```

7.7 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.

I just finished chapter 7 of Api on Rails tutorial by @kurenn!

Chapter 8

Placing Orders

Back in Chapter 7 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, because we will handle associations between 3 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 https://github.com/kurenn/market_place_api.git -b chapter7
```

Let's create a branch to start working:

```
$ git checkout -b chapter8
Switched to a new branch 'chapter8'
```

8.1 Modeling the order

If you recall Figure 2.1 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 on Section 8.3.

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
==  CreateOrders: migrating ===================================================
-- create_table(:orders)
   -> 0.0039s
==  CreateOrders: migrated (0.0040s) ==========================================
```

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

Listing 8.1: Order first specs (*spec/models/order_spec.rb*)

```ruby
require 'spec_helper'

describe Order do
  let(:order) { FactoryGirl.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:

```
class Order < ActiveRecord::Base
  belongs_to :user

  validates :total, presence: true,
                      numericality: { greater_than_or_equal_to: 0 }

  validates :user_id, presence: true
end
```

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

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

Finished in 0.04942 seconds
6 examples, 0 failures

Randomized with seed 62339
```

8.1.1 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 appropiate association.

Let's generate this model:

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

Let's migrate the database and prepare the test one:

```
$ rake db:migrate; rake db:test:prepare
==  CreatePlacements: migrating ===============================================
-- create_table(:placements)
   -> 0.0057s
==  CreatePlacements: migrated (0.0058s) ======================================
```

Let's add the order association specs first:

Listing 8.2: Order placements associations specs (*spec/models/order_spec.rb*)

```ruby
require 'spec_helper'

describe Order do
  let(:order) { FactoryGirl.build :order }
  subject { order }
  .
  .
  .
  it { should have_many(:placements) }
  it { should have_many(:products).through(:placements) }
end
```

The implementation is like so:

```ruby
class Order < ActiveRecord::Base
  .
  .
  .
  has_many :placements
  has_many :products, through: :placements
end
```

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

Listing 8.3: Product placements associations specs (*spec/models/product_spec.rb*)

```ruby
require 'spec_helper'

describe Product do
  .
  .
  .
  it { should have_many(:placements) }
  it { should have_many(:orders).through(:placements) }
  .
  .
  .
end
```

Let's add the code to make it pass:

```ruby
class Product < ActiveRecord::Base
  validates :title, :user_id, presence: true
  .
  .
  .
  has_many :placements
  has_many :orders, through: :placements
  .
  .
  .
end
```

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

Listing 8.4: Placement specs (*spec/models/placement_spec.rb*)

```ruby
require 'spec_helper'

describe Placement do
  let(:placement) { FactoryGirl.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.

```
class Placement < ActiveRecord::Base
  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:

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

Finished in 0.63099 seconds
47 examples, 0 failures

Randomized with seed 33160
```

Now that everything is nice and green, let's commit the changes and continue with Section 8.3

```
$ git add .
$ git commit -m "Associates products and orders with a placements model"
[chapter8 2bd331d] Associates products and orders with a placements model
 11 files changed, 105 insertions(+), 1 deletion(-)
 create mode 100644 app/models/order.rb
 create mode 100644 app/models/placement.rb
 create mode 100644 db/migrate/20140826170754_create_orders.rb
 create mode 100644 db/migrate/20140826173533_create_placements.rb
 create mode 100644 spec/factories/orders.rb
 create mode 100644 spec/factories/placements.rb
 create mode 100644 spec/models/order_spec.rb
 create mode 100644 spec/models/placement_spec.rb
```

8.2 User orders

We are just missing one little but very important part, which is to relate the user to the orders, which it already started as shown on Listing~{code:order_first_specs}, 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:

```
.
.
.
#line 20
it { should have_many(:products) }
it { should have_many(:orders) }
.
.
.
```

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

```
.
.
.
#line 9
has_many :products, dependent: :destroy
has_many :orders, dependent: :destroy
.
.
.
```

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

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

Finished in 0.06012 seconds
8 examples, 0 failures

Randomized with seed 22530
```

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

Finished in 0.21172 seconds
15 examples, 0 failures

Randomized with seed 49329
```

Let's commit this small changes and move on to section 8.3.

```
$ git add .
$ git commit -m 'Adds user order has many relation'
```

8.3 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
  create  app/controllers/api/v1/orders_controller.rb
  invoke  erb
  create    app/views/api/v1/orders
  invoke  rspec
  create    spec/controllers/api/v1/orders_controller_spec.rb
  invoke  assets
  invoke    coffee
  invoke    scss
```

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, and 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:

Listing 8.5: Orders controller index spec(*spec/controllers/api/v1/orders_controlller_spec.r*

```
describe Api::V1::OrdersController do

  describe "GET #index" do
    before(:each) do
      current_user = FactoryGirl.create :user
      api_authorization_header current_user.auth_token
      4.times { FactoryGirl.create :order, user: current_user }
      get :index, user_id: current_user.id
    end

    it "returns 4 order records from the user" do
      orders_response = json_response[:orders]
      expect(orders_response).to have(4).items
    end

    it { should respond_with 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:

```
.
.
.
#line 9
resources :users, :only => [:show, :create, :update, :destroy] do
  resources :products, :only => [:create, :update, :destroy]
  resources :orders, :only => [:index]
end
.
.
.
```

Now it is time for the orders controller implementation:

Listing 8.6: Orders controller index action(*spec/controllers/api/v1/orders_controlller.rb*)

```
class Api::V1::OrdersController < ApplicationController
  before_action :authenticate_with_token!
  respond_to :json

  def index
    respond_with current_user.orders
  end
end
```

And now all of our tests should pass:

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

Finished in 0.13456 seconds
2 examples, 0 failures

Randomized with seed 14135
```

We like our commits very atomic, so let's commit this changes:

```
$ git add .
$ git commit -m "Adds index orders controller action nested into the users resources"
```

8.3.1 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:

Listing 8.7: Orders controller show spec(*spec/controllers/api/v1/orders_controlller_spec.r*

```
.
.
```

```
.
#line 20
describe "GET #show" do
  before(:each) do
    current_user = FactoryGirl.create :user
    api_authorization_header current_user.auth_token
    @order = FactoryGirl.create :order, user: current_user
    get :show, user_id: current_user.id, id: @order.id
  end

  it "returns the user order record matching the id" do
    order_response = json_response[:order]
    expect(order_response[:id]).to eql @order.id
  end

  it { should respond_with 200 }
end
```

Let's add the implementation to make our tests pass:

On the **routes.rb** file add the show action to the orders resources:

```
resources :orders, :only => [:index, :show]
```

And the the implementation should look like this:

```
def show
  respond_with current_user.orders.find(params[:id])
end
```

Can you guess the result of running the tests now?.

If you expect that one test would fail, you are right, but do you know the reason, think about it, I'll wait.

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

Failures:

  1) Api::V1::OrdersController GET #show returns the user order record matching the id
     Failure/Error: expect(order_response[:id]).to eql @order.id
     NoMethodError:
       undefined method `[]' for nil:NilClass
     # ./spec/controllers/api/v1/orders_controller_spec.rb:31:in `block (3 levels) in <top (req

Finished in 0.19786 seconds
4 examples, 1 failure

Failed examples:

rspec ./spec/controllers/api/v1/orders_controller_spec.rb:29 # Api::V1::OrdersController GET #s

Randomized with seed 18274
```

Couln't wait?, well the answer is that we are expecting some format from the API, which is given by `ActiveModelSerializers`, which in this case we have not created an order serializer. This is fairly easy to fix:

```
$ rails g serializer order
```

And just by adding the serializer, our tests should be all green:

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

Finished in 0.17973 seconds
4 examples, 0 failures

Randomized with seed 35825
```

We will leave the order serializer as it is for now, but don't worry we will customize it later.

Let's commit the changes and move onto the create order action:

```
$ git add .
$ g commit -m "Adds the show action for order"
[chapter8 9038c1d] Adds the show action for order
 4 files changed, 23 insertions(+), 1 deletion(-)
 create mode 100644 app/serializers/order_serializer.rb
```

8.3.2 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:

Listing 8.8: Orders controller create spec(*spec/controllers/api/v1/orders_controlller_spec.rb*)

```
.
.
.
#line 36
describe "POST #create" do
  before(:each) do
    current_user = FactoryGirl.create :user
    api_authorization_header current_user.auth_token

    product_1 = FactoryGirl.create :product
    product_2 = FactoryGirl.create :product
    order_params = { total: 50, user_id: current_user.id, product_ids: [product_1.id, product_2
    post :create, user_id: current_user.id, order: order_params
  end

  it "returns the just user order record" do
    order_response = json_response[:order]
    expect(order_response[:id]).to be_present
  end
  it { should respond_with 201 }
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:

```
resources :orders, :only => [:index, :show, :create]
```

Then the implementation which is easy:

Listing 8.9: Orders controller create action(*spec/controllers/api/v1/orders_controlller.rb*)

```
.
.
.
#line 12
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
```

And now our tests should all be green:

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

Finished in 0.23719 seconds
```

```
6 examples, 0 failures

Randomized with seed 23197
```

Ok, so we have everything nice and green. We now should move on to the next chapter right?, well 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, and 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 simpliest 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 better said 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:

Listing 8.10: Order set_total!(*spec/models/order_spec.rb*)

```
.
.
.
#line 17
describe '#set_total!' do
  before(:each) do
    product_1 = FactoryGirl.create :product, price: 100
    product_2 = FactoryGirl.create :product, price: 85

    @order = FactoryGirl.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
```

We can now add the implementation:

Listing 8.11: Order set_total!(*app/models/order.rb*)

```
.
.
.
#line 17
def set_total!
  self.total = products.map(&:price).sum
end
```

Just before you run your tests, we need to update the `order` factory, just to make it more useful:

```
FactoryGirl.define do
  factory :order do
    user
    total 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.

```
before_validation :set_total!
```

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:

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

Finished in 0.17454 seconds
7 examples, 0 failures

Randomized with seed 3392u
```

This is now the moment to visit the `orders_controller_spec.rb` file and refactor some code:

Currently we have something like:

```
  describe "POST #create" do
    before(:each) do
      current_user = FactoryGirl.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryGirl.create :product
      product_2 = FactoryGirl.create :product
      order_params = { total: 50, user_id: current_user.id, product_ids: [product_1.id, product
      post :create, user_id: current_user.id, order: order_params
    end

    it "returns the just user order record" do
      order_response = json_response[:order]
      expect(order_response[:id]).to be_present
    end

    it { should respond_with 201 }
  end
```

We just need to remove the `user_id` and the `total` params as the user id is not really necessary and the total is being calculated through the model. After making the changes the code should look like:

```
describe "POST #create" do
  before(:each) do
    current_user = FactoryGirl.create :user
    api_authorization_header current_user.auth_token

    product_1 = FactoryGirl.create :product
    product_2 = FactoryGirl.create :product
    order_params = { product_ids: [product_1.id, product_2.id] }
    post :create, user_id: current_user.id, order: order_params
  end

  it "returns the just user order record" do
    order_response = json_response[:order]
    expect(order_response[:id]).to be_present
  end

  it { should respond_with 201 }
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:

```
def order_params
  params.require(:order).permit(:product_ids => [])
end
```

Your tests should still passing:

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

Finished in 0.24193 seconds
6 examples, 0 failures

Randomized with seed 48129
```

Let's commit the changes:

```
$ gg commit -m "Adds the create method for the orders controller"
[chapter8 ca47c29] Adds the create method for the orders controller
 6 files changed, 57 insertions(+), 8 deletions(-)
```

8.4 Customizing the Order json output

Now that we built the necessary endopoints for the orders, we can customize the information we want to render on the json output for each order.

To do this we can just open the `order_serializer.rb` file, which should look like:

```
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 everytime an order object is being parsed to json, so in this case I would say it is just fine:

Listing 8.12: Order show action spec(*spec/controllers/api/v1/orders_controller_spec.rb*)

```
describe "GET #show" do
  before(:each) do
    current_user = FactoryGirl.create :user
    api_authorization_header current_user.auth_token

    @product = FactoryGirl.create :product
    @order = FactoryGirl.create :order, user: current_user, product_ids: [@product.id]
    get :show, user_id: current_user.id, id: @order.id
  end
  .
  .
  .
  it "includes the total for the order" do
    order_response = json_response[:order]
    expect(order_response[:total]).to eql @order.total.to_s
  end

  it "includes the products on the order" do
    order_response = json_response[:order]
    expect(order_response[:products]).to have(1).item
  end
  .
  .
  .
end
```

By now we should have failing tests. But they are easy to fix on the order serializer

```
class OrderSerializer < ActiveModel::Serializer
  attributes :id, :total
  has_many :products
end
```

And now all of our tests should be green:

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

Finished in 0.38289 seconds
8 examples, 0 failures

Randomized with seed 44243
```

If you recall Chapter 7 on Box 7.2 we embeded the user into the product, in order to retrieve some information, but in this 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:

```
class OrderProductSerializer < ProductSerializer
end
```

This will keep rendered data on sync, and now to remove the embebed user we simply add the following method on the gem documentation. For more information visit ActiveModelSerializer:

```
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`:

```
class OrderSerializer < ActiveModel::Serializer
  attributes :id, :total
  has_many :products, serializer: OrderProductSerializer
end
```

And our tests should still passing:

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

Finished in 0.34248 seconds
8 examples, 0 failures

Randomized with seed 54262
```

Let’s commit this and move onto the next section:

```
$ git commit -m "Adds a custom order product serializer to remove the user association"
[chapter8 ecac23c] Adds a custom order product serializer to remove the user association
 3 files changed, 20 insertions(+), 2 deletions(-)
 create mode 100644 app/serializers/order_product_serializer.rb
```

8.5 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 9, 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 g 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`

```
#line 37
group :test do
  gem "rspec-rails", "~> 2.14"
  gem "shoulda-matchers"
  gem "email_spec"
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 https://github.com/bmabey/email-spec#rspec.

When you are done, your `spec_helper.rb` file should look like:

```
.
.
.
require "email_spec"
.
.
.
RSpec.configure do |config|
.
.
.
config.include(EmailSpec::Helpers)
config.include(EmailSpec::Matchers)
.
.
.
end
```

Now we can add some tests for the order mailer we created earlier:

Listing 8.13: Order mailer spec(*spec/mailers/order_mailer_spec.rb*)

```
require "spec_helper"

describe OrderMailer do
   include Rails.application.routes.url_helpers

  describe ".send_confirmation" do
    before(:all) do
      @order = FactoryGirl.create :order
      @user = @order.user
      @order_mailer = OrderMailer.send_confirmation(@order)
    end

    it "should be set to be delivered to the user from the order passed in" do
      @order_mailer.should deliver_to(@user.email)
    end

    it "should be set to be send from no-reply@marketplace.com" do
      @order_mailer.should deliver_from('no-reply@marketplace.com')
    end

    it "should contain the user's message in the mail body" do
      @order_mailer.should have_body_text(/Order: ##{@order.id}/)
    end

    it "should have the correct subject" do
      @order_mailer.should have_subject(/Order Confirmation/)
    end

    it "should have the products count" do
      @order_mailer.should 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:

Listing 8.14: Order mailer(*app/mailers/order_mailer.rb*)

```
class OrderMailer < ActionMailer::Base
  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.

```
$ touch app/views/order_mailer/send_confirmation.html.erb
$ touch app/views/order_mailer/send_confirmation.txt.erb
```

The html version looks like this:

```
<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>
```

And the text version like:

```
Order: #<%= @order.id %>

You ordered <%= @order.products.count %> products:

<% @order.products.each do |product| %>
  <%= product.title %> - <%= number_to_currency product.price %>
<% end %>
```

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

```
$ bundle exec rspec spec/mailers/order_mailer_spec.rb
.....

Finished in 0.13474 seconds
5 examples, 0 failures

Randomized with seed 20538
```

We just need to call the `send_confirmation` method into the create action on the orders controller:

```
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
```

To make sure we did not break anything on the orders, we can just run the specs from the orders controller:

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

Finished in 0.40412 seconds
8 examples, 0 failures

Randomized with seed 19910
```

Let's finish this section by commiting this:

```
$ git commit -am "Adds order confirmation mailer"
[chapter8 91063df] Adds order confirmation mailer
 8 files changed, 73 insertions(+), 3 deletions(-)
 create mode 100644 app/mailers/order_mailer.rb
 create mode 100644 app/views/order_mailer/send_confirmation.html.erb
 create mode 100644 app/views/order_mailer/send_confirmation.txt.erb
 create mode 100644 spec/mailers/order_mailer_spec.rb
```

8.6 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.

Show me some love on twitter:

I just finished chapter 8 of Api on Rails tutorial by @kurenn!

After chapter 9, we will focus on optimization, pagination and some other cool stuff that will definitely help you build a better app.

Chapter 9

Improving orders

Back in Chapter 8 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/kurenn/market_place_api.git -b chapter8
```

Let's create a branch to start working:

```
$ git checkout -b chapter9
Switched to a new branch 'chapter9'
```

9.1 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 g migration add_quantity_to_products quantity:integer
      invoke  active_record
      create    db/migrate/20150105212608_add_quantity_to_products.rb
```

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:

```
class AddQuantityToProducts < ActiveRecord::Migration
  def change
    add_column :products, :quantity, :integer, default: 0
  end
end
```

Migrate the database and prepare the test one:

```
$ bundle exec rake db:migrate && bundle exec rake db:test:prepare
```

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. If you recall Listing 8.9 we are expecting an array of product ids. I'm going to try keep things simple and will send a 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:

```
# The first position for the inner arrays is the product id and
# the second the quantity to buy
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:

```
require 'spec_helper'

describe Order do
  .
  .
  .
  describe '#set_total!' do
          ...
  end

  describe "#build_placements_with_product_ids_and_quantities" do
    before(:each) do
      product_1 = FactoryGirl.create :product, price: 100, quantity: 5
      product_2 = FactoryGirl.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
    end
  end

end
```

Then into the implementation:

```
class Order < ActiveRecord::Base
  .
  .
  .
  def set_total!
          ...
  end

  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]

      self.placements.build(product_id: id)
    end
  end
end
```

And if we run our tests, they should be all nice and green:

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

Finished in 0.21348 seconds
9 examples, 0 failures

Randomized with seed 45644
```

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 commiting this is to update the `orders_controller_spec` along with its implementation.

First we update the orders_controller_spec file:

```
require 'spec_helper'

describe Api::V1::OrdersController do
  .
  .
  .
  describe "POST #create" do
    before(:each) do
      current_user = FactoryGirl.create :user
      api_authorization_header current_user.auth_token

      product_1 = FactoryGirl.create :product
      product_2 = FactoryGirl.create :product
      order_params = { product_ids_and_quantities: [[product_1.id, 2],[ product_2.id, 3]] }
      post :create, user_id: current_user.id, order: order_params
    end

    it "returns just user order record" do
      order_response = json_response[:order]
      expect(order_response[:id]).to be_present
    end

    it "embeds the two product objects related to the order" do
      order_response = json_response[:order]
      expect(order_response[:products].size).to eql 2
    end

    it { should respond_with 201 }
  end

end
```

Then we need to update the `orders_controller`:

```
class Api::V1::OrdersController < ApplicationController
  .
  .
  .
  def create
    order = current_user.orders.build
    order.build_placements_with_product_ids_and_quantities(params[:order][:product_ids_and_quan

    if order.save
      order.reload #we reload the object so the response displays the product objects
      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
```

Notice 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.

```
FactoryGirl.define do
  factory :product do
    title { FFaker::Product.product_name }
    price { rand() * 100 }
    published false
    user
    quantity 5 #this is the line we added
  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 g migration add_quantity_to_placements quantity:integer
     invoke  active_record
     create    db/migrate/20150105234750_add_quantity_to_placements.rb
```

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:

```
class AddQuantityToPlacements < ActiveRecord::Migration
  def change
    add_column :placements, :quantity, :integer, default: 0
  end
end
```

Run the migrations and prepare the test database:

```
$ bundle exec rake db:migrate && bundle exec rake db:test:prepare
==  AddQuantityToPlacements: migrating ========================================
-- add_column(:placements, :quantity, :integer, {:default=>0})
   -> 0.0092s
==  AddQuantityToPlacements: migrated (0.0093s) ===============================
```

Let's document the `quantity` attribute through a unit test like so:

```
require 'spec_helper'

describe Placement do
  .
  .
  .
  it { should respond_to :product_id }
  it { should respond_to :quantity }
  .
  .
  .
end
```

Let's make sure everything is green:

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

Finished in 0.02985 seconds
5 examples, 0 failures
```

Now we just need to update the `build_placements_with_product_ids_and_qua` to add the `quantity` for the placements:

Listing 9.1: Build placements with product ids and quantities(*app/models/order.rb*)

```
class Order < ActiveRecord::Base
.
.
.
#line 15
  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]

      self.placements.build(product_id: id, quantity: quantity)
    end
  end
end
```

Our `order_spec.rb` should be still green:

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

Finished in 0.18384 seconds
8 examples, 0 failures

Randomized with seed 2756
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Adds quantity to placements"
```

9.1.1 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:

Listing 9.2: Placements factory (*spec/factories/placements.rb*)

```
FactoryGirl.define do
  factory :placement do
    order
    product
    quantity 1
  end

end
```

And then we can simply add some specs:

Listing 9.3: Decrease product quantity method spec(*spec/models/placement*$_s$*pec.rb*)

```
require 'spec_helper'

describe Placement 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.q
    end
  end
end
```

The implementation is fairly easy as shown on Listing 9.4.

Listing 9.4: Decrease product quantity method (*app/models/placement.rb*)

```
class Placement < ActiveRecord::Base
  .
  .
  .
  #line 5
  def decrement_product_quantity!
    self.product.decrement!(:quantity, quantity)
  end
end
```

And now we just have to hook this method into an `after_create` callback and we should be good to go:

```
class Placement < ActiveRecord::Base
  .
  .
  .
  after_create :decrement_product_quantity!

  def decrement_product_quantity!
    self.product.decrement!(:quantity, quantity)
  end
end
```

We just now need to make sure that there are enough products on stock to create a placement record, but first it would be a good idea to commit the changes:

```
$ git add .
$ git commit -m "Decrements the product quantity by the placement quantity"
```

9.2 Validation for product on stock

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 thorugh 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
```

Then we create a file for the validator:

```
$ 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.

Listing 9.5: Order enough products in stock validation (*spec/models/order_spec.rb*)

```
require 'spec_helper'

describe Order do
  .
  .
  .
  describe "#build_placements_with_product_ids_and_quantities" do
  end

  describe "#valid?" do
    before do
      product_1 = FactoryGirl.create :product, price: 100, quantity: 5
      product_2 = FactoryGirl.create :product, price: 85, quantity: 10


      placement_1 = FactoryGirl.build :placement, product: product_1, quantity: 3
      placement_2 = FactoryGirl.build :placement, product: product_2, quantity: 15

      @order = FactoryGirl.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:

```
class EnoughProductsValidator < ActiveModel::Validator
  def validate(record)
    record.placements.each do |placement|
      product = placement.product
      if placement.quantity > product.quantity
        record.errors["#{product.title}"] << "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:

```
class Order < ActiveRecord::Base
  belongs_to :user

  validates :user_id, presence: true
  validates_with EnoughProductsValidator #this is the line we added for the custom validator
  .
  .
  .
end
```

And now if you run your tests, everything should be nice and green:

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

Finished in 0.23009 seconds
9 examples, 0 failures

Randomized with seed 454
```

Let's commit the changes:

```
$ git add .
$ git commit -m "Adds validator for order with not enough products on stock"
[chapter9 7bd9db4] Adds validator for order with not enough products on stock
 3 files changed, 31 insertions(+)
 create mode 100644 app/validators/enough_products_validator.rb
```

9.3 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:

```
def set_total!
  self.total = products.map(&:price).sum
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:

```
  describe '#set_total!' do
    before(:each) do
      product_1 = FactoryGirl.create :product, price: 100
      product_2 = FactoryGirl.create :product, price: 85

      placement_1 = FactoryGirl.build :placement, product: product_1, quantity: 3
      placement_2 = FactoryGirl.build :placement, product: product_2, quantity: 15

      @order = FactoryGirl.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
```

And the implementation is fairly easy:

```
  def set_total!
    self.total = 0
    placements.each do |placement|
      self.total += placement.product.price * placement.quantity
    end
  end
```

And the specs should be green:

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

Finished in 0.23888 seconds
9 examples, 0 failures

Randomized with seed 13568
```

Let's commit the changes and wrap up.

```
$ git commit -am "Updates the total calculation for order"
[chapter9 1f4891a] Updates the total calculation for order
 2 files changed, 7 insertions(+), 2 deletions(-)
```

9.4 Conclusion

Oh you are here!, let me congratulate you, it's been a long way since chapter 1, but you are 1 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`, `caching` and `background jobs`, so buckle up, it is going to be a bumpy ride.

Show me some love on twitter:

I just finished chapter 9 of Api on Rails tutorial by @kurenn!

Chapter 10

Optimization

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 Chapter 9 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:

- Pagination
- Background Jobs
- 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/kurenn/market_place_api.git -b chapter9
```

Let's now create a branch to start working:

```
$ cd market_place_api/
$ git checkout -b chapter10
Switched to a new branch 'chapter10'
```

10.1 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 wheter 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 Chapter 1 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.

10.1.1 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`:

```
gem 'kaminari'
```

And then run the bundle command to install it:

```
$ bundle install
```

Now we can go to the **`index`** action on the **`products_controller`** and add the pagination methods as pointed on the documentation:

```
def index
  respond_with Product.search(params).page(params[:page]).per(params[:per_page])
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:

```
"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:

Listing 10.1: Products paginated(*spec/controllers/api/v1/products_controller_spec.rb*)

```
require 'spec_helper'

describe Api::V1::ProductsController do

  describe "GET #show" do
  ...
  end
  .
  .
  .
  describe "GET #index" do
```

```
    before(:each) do
      4.times { FactoryGirl.create :product }
    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
        products_response = json_response
        expect(products_response[:products]).to have(4).items
      end

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

      # we added this lines for the pagination
      it { expect(json_response).to have_key(:meta) }
      it { expect(json_response[:meta]).to have_key(:pagination) }
      it { expect(json_response[:meta][:pagination]).to have_key(:per_page) }
      it { expect(json_response[:meta][:pagination]).to have_key(:total_pages) }
      it { expect(json_response[:meta][:pagination]).to have_key(:total_objects) }

      it { should respond_with 200 }
    end

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

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

We should have 6 failing tests by now, yes you read it right 6, although we just added 5, which means we broke something else.

```
$ bundle exec rspec spec/controllers/api/v1/products_controller_spec.rb
.
.
.
Finished in 0.71762 seconds
23 examples, 6 failures
```

If you are wondering which test I'm talking about here it is:

```
1) Api::V1::ProductsController GET #index when product_ids parameter is sent returns just the p
     Failure/Error: get :index, product_ids: @user.product_ids
     NoMethodError:
       undefined method `page' for #<Array:0x007fabb761a818>
     # ./app/controllers/api/v1/products_controller.rb:8:in `index'
     # ./spec/controllers/api/v1/products_controller_spec.rb:59:in `block (4 levels) in <top (r
```

The error is actually on the `search` method of the `Product` model, because for some reason kaminari is expecting an active record relation instead of an array. It is actually really easy to fix:

app/models/product.rb

```
  def self.search(params = {})
    - products = params[:product_ids].present? ? Product.find(params[:product_ids]) : Product.a
    + products = params[:product_ids].present? ? Product.where(id: params[:product_ids]) : Prod
    .
    .
    .
  end
```

Did you notice the change?, if not let me tell you what changed. Instead of fetching the record using the `find` method with the product_ids params, I just change it to a `where` clause which returns an ActiveRecord::Relation, exactly what we need.

Now if we run the specs again, that spec should be green:

```
$ bundle exec rspec spec/controllers/api/v1/products_controller_spec.rb
.
.
.
Finished in 0.88109 seconds
23 examples, 5 failures
```

Now that we fixed that, let's add the pagination information, we need to do it on the `products_controller.rb` file:

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

  def index
    products = Product.search(params).page(params[:page]).per(params[:per_page])
    render json: products, meta: { pagination:
                                   { per_page: params[:per_page],
                                     total_pages: products.total_pages,
                                     total_objects: products.total_count } }
  end
  .
  .
  .
end
```

As you may already notice, I changed the `respond_with` for a `render` call, just to make a custom json output, in this case the pagination information.

Now if we run the specs, they should be all passing:

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

Finished in 1.28 seconds
23 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 commit -m "Adds pagination for the products index action to optimize response"
[chapter10 6716515] Adds pagination for the products index action to optimize response
 5 files changed, 18 insertions(+), 4 deletions(-)
```

10.1.2 Orders

Now it is time to do exactly the same for the `orders` list endpoint, which by now should be really easy to implement. But first, let's add some specs to the `orders_controller_spec.rb` file:

```
require 'spec_helper'

describe Api::V1::OrdersController do

  describe "GET #index" do
    before(:each) do
      current_user = FactoryGirl.create :user
      api_authorization_header current_user.auth_token
      4.times { FactoryGirl.create :order, user: current_user }
      get :index, user_id: current_user.id
    end

    it "returns 4 order records from the user" do
      orders_response = json_response[:orders]
      expect(orders_response).to have(4).items
    end

    # These lines are the ones added to test the pagination
    it { expect(json_response).to have_key(:meta) }
    it { expect(json_response[:meta]).to have_key(:pagination) }
    it { expect(json_response[:meta][:pagination]).to have_key(:per_page) }
    it { expect(json_response[:meta][:pagination]).to have_key(:total_pages) }
    it { expect(json_response[:meta][:pagination]).to have_key(:total_objects) }

    it { should respond_with 200 }
  end
  .
  .
  .
end
```

And as you may already wonder, we should have 5 tests failing:

```
$ bundle exec rspec spec/controllers/api/v1/orders_controller_spec.rb
.
.
.
Finished in 0.59358 seconds
14 examples, 5 failures
```

Let's turn the red into green:

```
class Api::V1::OrdersController < ApplicationController
  before_action :authenticate_with_token!
  respond_to :json

  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:

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

Finished in 0.56828 seconds
14 examples, 0 failures

Randomized with seed 6870
```

Let's place and commit, because a refactor is coming:

```
$ git commit -m "Adds pagination for orders index action"
[chapter10 3f06f99] Adds pagination for orders index action
 2 files changed, 11 insertions(+), 1 deletion(-)
```

10.1.3 Refactoring pagination

If you have been following this tutorial or if you are an experienced Rails developer, you probably love keep things DRY, and although the code we generated for pagination is not to much, I think it would be a good practice to clean no just the implementation but also the specs as bothof them have duplication.

We are going to first clean the specs, which in this case are:

```
it { expect(json_response).to have_key(:meta) }
it { expect(json_response[:meta]).to have_key(:pagination) }
it { expect(json_response[:meta][:pagination]).to have_key(:per_page) }
it { expect(json_response[:meta][:pagination]).to have_key(:total_pages) }
it { expect(json_response[:meta][:pagination]).to have_key(:total_objects) }
```

Let's add a `shared_examples` folder under the `spec/support/` directory:

```
$ mkdir spec/support/shared_examples
```

And then create a file for the pagination example:

```
$ touch spec/support/shared_examples/pagination.rb
```

And on the `pagination.rb` file you can just add the following lines:

```
shared_examples "paginated list" do
  it { expect(json_response).to have_key(:meta) }
  it { expect(json_response[:meta]).to have_key(:pagination) }
  it { expect(json_response[:meta][:pagination]).to have_key(:per_page) }
  it { expect(json_response[:meta][:pagination]).to have_key(:total_pages) }
  it { expect(json_response[:meta][:pagination]).to have_key(:total_objects) }
end
```

This shared example can now be use as a substitude for the 5 tests on the `orders_controller_spec.rb` and `products_controller_spec.rb` files like so:

spec/controllers/api/v1/orders_controller_spec.rb

```
it_behaves_like "paginated list"
```

spec/controllers/api/v1/products_controller_spec.rb

```
it_behaves_like "paginated list"
```

And both specs should be passing.

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.

Listing 10.2: Application controller with pagination method(*app/controllers/application_controller.rb*)

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

  include Authenticable

  protected

    def pagination(paginated_array, per_page)
      { pagination: { per_page: per_page.to_i,
                      total_pages: paginated_array.total_pages,
                      total_objects: paginated_array.total_count } }
    end
end
```

And now we can substitude the pagination hash on both controllers for the method, like so:

app/controllers/api/v1/orders_controller.rb

```
  def index
    orders = current_user.orders.page(params[:page]).per(params[:per_page])
    render json: orders, meta: pagination(orders, params[:per_page])
  end
```

app/controllers/api/v1/products_controller.rb

```
  def index
    products = Product.search(params).page(params[:page]).per(params[:per_page])
    render json: products, meta: pagination(products, params[:per_page])
  end
```

If you run the specs for each file they should be all nice and green:

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

Finished in 0.56902 seconds
14 examples, 0 failures

Randomized with seed 61566
```

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

Finished in 0.73919 seconds
23 examples, 0 failures

Randomized with seed 35453
```

This would be a good moment to commit the changes and move on to the next Section 10.2 about background jobs.

```
$ git commit -m "Refactors the pagination behavior for index actions"
[chapter10 64d4c5e] Refactors the pagination behavior for index actions
 6 files changed, 19 insertions(+), 18 deletions(-)
 create mode 100644 spec/support/shared_examples/pagination.rb
```

10.2 Background Jobs

When it comes to optimization strategies, you cannot miss adding a background jobs gem for long processes which could lead to a bad experience or timeout errors all over the place. Our application is fairly simple and actually the only thing that could steal some CPU time is sending mails, which in our case, we are just sending a confirmation for the order.

Although it won't take much I find this relevant for future long-time processes and allow the server not to lock. There are so many options out there for handling this, Sidekiq, Resque, or Delayed Job, we will be using the last one, just to keep things simple.

We first add the gem to the `Gemfile`:

```
gem 'delayed_job_active_record'
```

And then run the `bundle` command to install it along with the `delayed_job` generate method:

```
$ bundle install
$ rails generate delayed_job:active_record
$ rake db:migrate
```

This will generate a table for delayed jobs where are going to be enque for later processing. For more information head to the documentation

And now that we have eveything setup, we just need to update the line where we send the email after the order has been placed, like so:

app/controllers/api/v1/orders_controller.rb

```
  .
  .
  .
  def create
    order = current_user.orders.build
    order.build_placements_with_product_ids_and_quantities(params[:order][:product_ids_and_quan

    if order.save
      order.reload #we reload the object so the response displays the product objects
      OrderMailer.delay.send_confirmation(order) #this is the line
      render json: order, status: 201, location: [:api, current_user, order]
    else
      render json: { errors: order.errors }, status: 422
    end
  end
  .
  .
  .
```

As you can see we update the `OrderMailer.send_confirmation(order).delive` to `OrderMailer.delay.send_confirmation(order)` as suggested by the documentation.

If you want to actually perform the delivery in your development environment, you just need to add a couple of lines to the `development.rb` file:

```
MarketPlaceApi::Application.configure do
  .
  .
  .
  config.action_mailer.delivery_method = :smtp
  config.action_mailer.smtp_settings = { :address => "localhost", :port => 1025 }

  config.action_mailer.default_url_options = { :host => "localhost",
                                               only_path: false }
end
```

Let’s commit and move onto the next Section 10.3:

```
$ git commit -m "adds delayed jobs for long task process"
[chapter10 4be4964] adds delayed jobs for long task process
 7 files changed, 58 insertions(+), 2 deletions(-)
 create mode 100755 bin/delayed_job
 create mode 100644 db/migrate/20150116223258_create_delayed_jobs.rb
```

10.3 API Caching

There is currently an implementation for this with the active_model_serializers gem which is really easy to handle. Although in older versions of the gem, this implementation may change, it gets the job done.

If we perform a request to the `products` list we will notice that the response time takes about `0.14` miliseconds using sabisu, but if we add just a few lines to the `ProductSerializer` class as shown on Listing 10.3, we will see an improvement of about 90% of the response time, as you can see on Figure 10.1 vs Figure 10.2

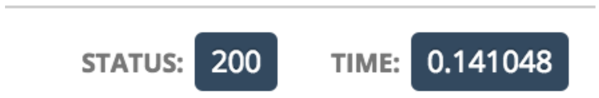

STATUS: 200 TIME: 0.141048

Figure 10.1: Response Time without caching

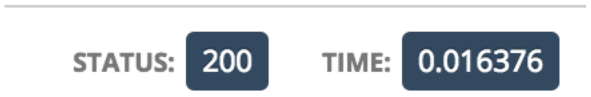

STATUS: 200 TIME: 0.016376

Figure 10.2: Response Time with caching

Listing 10.3: Product serializer with caching (*app/serializers/product$_serializer.rb$*)

```
class ProductSerializer < ActiveModel::Serializer
  cached

  attributes :id, :title, :price, :published
  has_one :user

  def cache_key
    [object, scope]
  end
end
```

As you can see the implementation is fairly easy and the benefits are huge, but be aware for any further change on the caching implementation that gem may implement. I'll add the `cached` and `cache_key` methods on every serializer just for demonstration purposes.

On a real environment you probably want to handle this with care.

Let's commit the changes and wrap up:

```
$ git commit -m "Adds caching for the serializers"
[chapter10 c3378b8] Adds caching for the serializers
 4 files changed, 24 insertions(+)
```

10.4 Conclusion

If you reach this point that means you are done with the book, good job!, you just become a great API Rails developer that's for sure.

Thanks for taking this great adventure all along with me, I hope you enjoyed the travel as much as I did. We should take a beer sometime.

Show me some love on twitter:

I'm a Rails API rockstar thanks to @kurenn!