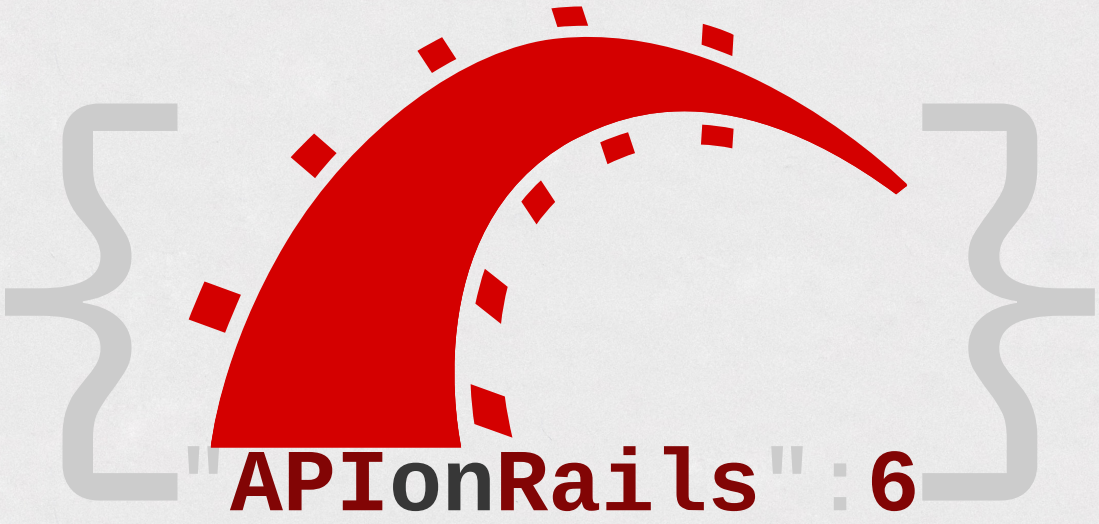
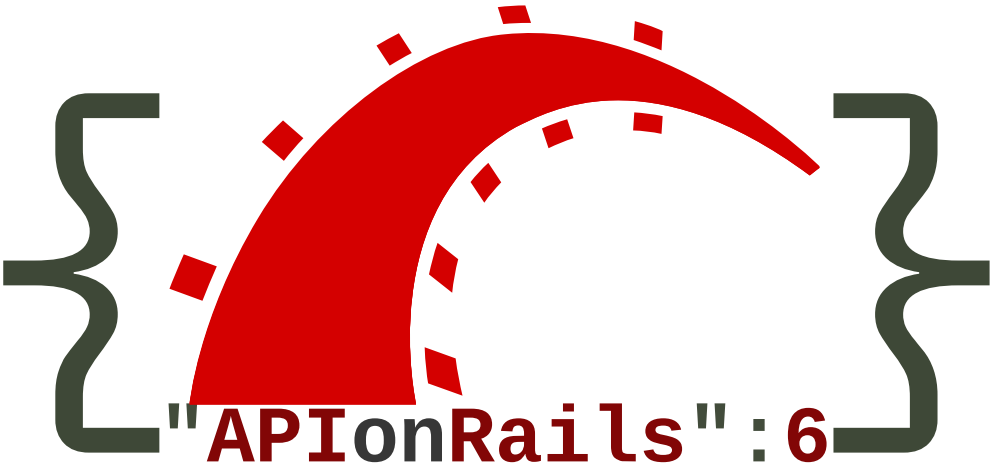


Alexandre Rousseau



API on Rails 6

Alexandre Rousseau

Version 6.9, 2020-09-01

Table of Contents

Avant-propos

"API on Rails 6" est basé sur "APIs on Rails: Building REST APIs with Rails". Celui-ci fut initialement publié en 2014 par Abraham Kuri sous les licences MIT et Beerware.

Cette première version, non maintenue, était prévue pour la version 4 de Ruby on Rails qui ne reçoit plus que les mises à jour de sécurité. J'ai voulu mettre à jour cet excellent ouvrage et contribuer à la communauté francophone en le traduisant moi-même et en l'adaptant aux nouvelles versions de Ruby on Rails. Ce livre est donc disponible pour la version 5.2 de Ruby on Rail et la version 6.0 (celle que vous lisez actuellement).

> **NOTE** Ce livre est aussi disponible dans la langue de Shakespeare.

Dans cette version je n'ai pas seulement cherché à rendre les exemples compatibles mais j'ai aussi voulu simplifier la mise en place d'une API avec Rails en utilisant des librairies plus récentes et surtout: maintenues.

A propos de l’auteur original

Je m’appelle Alexandre Rousseau et je suis un développeur Rails passionné avec plus de 4 ans d’expérience (à l’heure où j’écris ces lignes). Je suis actuellement associé chez (iSignif SAS) où je construis et maintiens un produit de type SAAS en utilisant Rails. Je contribue aussi à la communauté Ruby en produisant et maintenant quelques gemmes que vous pouvez consulter sur mon profil Rubygems.org. La plupart de mes projets sont sur GitHub donc n’hésitez pas à me suivre.

Tout le code source de ce livre est disponible au format Asciidoctor sur GitHub. Ainsi n’hésitez pas à forker le projet si vous voulez l’améliorer ou corriger une faute qui m’aurait échappée.

Droits d’auteur et licence

Merci

Un grand merci à tous les contributeurs de Github qui rendent ce livre vivant. Par ordre alphabétique :

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

Introduction

Bienvenue sur API on Rails 6, un tutoriel sous stéroïdes pour apprendre la meilleure façon de construire votre prochaine API avec Rails. Le but de ce livre est de vous fournir une méthodologie complète pour développer une API RESTful en suivant les meilleures pratiques.

Lorsque vous en aurez fini avec ce livre, vous serez en mesure de créer votre propre API et de l'intégrer à n'importe quel client comme un navigateur Web ou une application mobile. Le code généré est construit avec Ruby on Rails 6.0 qui est la version actuelle.

L'intention de ce livre n'est pas seulement de vous apprendre à construire une API avec Rails mais plutôt de vous apprendre comment construire une API **évolutive** et **maintenable** avec Rails. C'est-à-dire améliorer vos connaissances actuelles avec Rails. Dans ce voyage, vous allez apprendre à:

- Utiliser Git pour le contrôle de version
- Construire des réponses JSON
- Tester vos points d'entrées avec des tests unitaires et fonctionnels
- Mettre en place une authentification avec des JSON Web Tokens (JWT)
- Utiliser les spécifications JSON:API
- Optimiser et mettre en cache l'API

Je vous recommande fortement de suivre toutes les étapes de ce livre. Essayez de ne pas sauter des chapitres car je vais vous proposer des conseils et des astuces pour vous améliorer tout au long du livre. Vous pouvez vous considérer comme le personnage principal d'un jeu vidéo qui obtient un niveau supérieur à chaque chapitre.

Dans ce premier chapitre je vous expliquerai comment configurer votre environnement (au cas où vous ne l'auriez pas déjà fait). Nous allons ensuite créer une application appelée `market_place_api`. Je veillerai à vous enseigner les meilleures pratiques que j'ai pu apprendre au cours de mon expérience. Cela signifie qu'après avoir initialisé le projet, nous commencerons à utiliser **Git**.

Dans les prochains chapitres, nous allons construire l'application en suivant une méthode de travail simple que j'utilise quotidiennement. Nous développerons toute l'application en utilisant le **développement piloté par les tests** (TDD). Je vous expliquerai aussi l'intérêt d'utiliser une API pour votre prochain projet et de choisir un format de réponse adapté comme le JSON ou le XML. Plus loin, nous mettrons

les mains dans le code et nous compléterons les bases de l'application en construisant toutes les routes nécessaires. Nous sécuriserons aussi l'accès à l'API en construisant une authentification par échange d'en-têtes HTTP. Enfin, dans le dernier chapitre, nous ajouterons quelques techniques d'optimisation pour améliorer la structure et les temps de réponse du serveur.

L'application finale sera une application de place de marché qui permettra à des vendeurs de mettre en place leur propre boutique en ligne. Les utilisateurs seront en mesure de passer des commandes, télécharger des produits et plus encore. Il existe de nombreuses options pour créer une boutique en ligne comme Shopify, Spree ou Magento.

Tout au long de ce voyage (cela dépend de votre expertise), vous allez vous améliorer et être en mesure de mieux comprendre certaines des meilleures ressources Rails. J'ai aussi pris certaines des pratiques que j'ai trouvées sur Railscasts, CodeSchool et JSON API.

Conventions sur ce livre

Les conventions de ce livre sont basées sur celles du Tutoriel Ruby on Rails. Dans cette section, je vais en mentionner quelques-unes que vous ne connaissez peut-être pas.

Je vais utiliser de nombreux exemples en utilisant des ligne de commande. Je ne vais pas traiter avec Windows cmd (désolé les gars). Je vais baser tous les exemples en utilisant l'invite de ligne de commande de style Unix. Voici un exemple:

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

J'utiliserai quelques principes spécifiques à Ruby. C'est-à-dire:

- *"Éviter"* signifie que vous n'êtes pas censé le faire.
- *"Préférer"* indique que parmi les deux options, la première est la plus appropriée.
- *"Utiliser"* signifie que vous êtes en mesure d'utiliser la ressource.

Si vous rencontrez une erreur quelconque lors de l'exécution d'une commande, je vous recommande d'utiliser votre moteur de recherche pour trouver votre solution. Malheureusement, je ne peux pas couvrir toutes les erreurs possibles. Si vous rencontrez des problèmes avec ce tutoriel, vous pouvez toujours m'envoyer un email.

Environnements de développement

Pour presque tous les développeurs, l'une des parties les plus douloureuses est de mettre en place un environnement de développement confortable. Si vous le faites correctement, les prochaines étapes devraient être un jeu d'enfant. Je vais vous guider dans cette étape afin de vous faciliter la tâche et de vous motiver.

Éditeurs de texte et Terminal

Les environnements de développement diffèrent d'un ordinateur à l'autre. Ce n'est pas le cas avec les éditeurs de texte. Je pense que pour le développement avec Rails, un IDE est beaucoup trop lourd. Cependant certains pensent que c'est la meilleure façon de travailler. Si c'est votre cas, je vous recommande d'essayer RadRails ou RubyMine. Tous deux sont bien maintenus et possèdent de nombreuses intégrations par défaut. Maintenant, pour ceux qui, comme moi, préfèrent des outils simples, je peux vous dire qu'il existe beaucoup d'outils disponibles que vous pourrez personnaliser via des plugins et plus.

- **Éditeur de texte**: J'utilise personnellement Vim comme éditeur. Au cas où vous n'êtes pas un fan de Vim, il y a beaucoup d'autres solutions comme Sublime Text qui est facile à prendre en main et surtout multi-plateforme . Il est fortement inspiré par TextMate. Une troisième option est d'utiliser un éditeur de texte plus récent comme Atom de GitHub. C'est un éditeur de texte prometteur fait en JavaScript. Il est facile à personnaliser pour répondre à vos besoins. N'importe lequel des éditeurs que je viens de vous présenter fera le travail. Choisissez donc celui où vous êtes le plus à l'aise.
- **Terminal**: Je ne suis pas un fan de l'application Terminal par défaut sous Mac OS. Je recommande iTerm2, qui est un remplacement de terminal pour Mac OS. Si vous êtes sous Linux, vous avez probablement déjà un bon terminal.

Navigateur web

Quand au navigateur, je conseillerai directement Firefox. Mais d'autres développeurs utilisent Chrome ou même Safari. N'importe lequel d'entre eux vous aidera à construire l'application que vous voulez. Ils proposent tous un bon inspecteur pour le DOM, un analyseur de réseau et de nombreuses autres fonctionnalités que vous connaissez peut-être déjà.

Gestionnaire de paquets

- **Mac OS**: Il existe de nombreuses options pour gérer la façon dont vous installez les paquets sur votre Mac, comme Mac Ports ou Homebrew. Les deux sont de bonnes options, mais je choisirais la dernière. J'ai rencontré moins de problèmes lors de l'installation de logiciels avec Homebrew. Pour installer brew il suffit d'exécuter la commande ci-dessous:

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

- **Linux**: Vous êtes déjà prêts! Peu importe si vous utilisez apt, pacman, yum tant que vous vous sentez à l'aise et que vous savez comment installer des paquets.

Git

Nous utiliserons beaucoup Git et vous devriez aussi l'utiliser (non seulement pour ce tutoriel mais aussi pour tous vos projets). Pour l'installer, c'est très facile:

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

Ruby

Il existe de nombreuses façons d'installer et de gérer Ruby. Vous devriez probablement déjà avoir une version installée sur votre système. Pour connaître votre version, tapez simplement:

```
$ ruby -v
```

Rails 6.0 nécessite l'installation de la version 2.5 ou supérieure.

Pour l'installer, je vous recommande d'utiliser Ruby Version Manager (RVM) ou rbenv.

Le principe de ces outils est de permettre d'installer plusieurs versions de Ruby sur une même machine, dans un environnement hermétique à une éventuelle version installée sur votre système d'exploitation et de pouvoir basculer de l'une à l'autre facilement.

Dans ce tutoriel, nous allons utiliser RVM mais peu importe laquelle de ces deux options vous utiliserez.

Pour installer RVM, rendez vous sur https://rvm.io/ et installez la clé GPG [1]. Une fois cela fait:

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

Ensuite, vous pouvez installer la dernière version de Ruby:

```
$ rvm install 2.6
```

Si tout s'est bien passé, il est temps d'installer le reste des dépendances que nous allons utiliser.

Gemmes, Rails et bibliothèques manquantes

Tout d'abord, nous mettons à jour les Gemmes sur l'ensemble du système:

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

Dans la plupart des cas, si vous êtes sous Mac OS, vous devriez installer des bibliothèques supplémentaires:

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

Nous installons ensuite les gemmes nécessaires et ignorons la documentation pour chaque gemme:

```
$ printf 'gem: --no-document' >> ~/.gemrc
$ gem install bundler
$ gem install foreman
$ gem install rails -v 6.0.0.rc1
```

NOTE	Vous pouvez vous demander ce que signifie RC1. RC1 signifie *Release Candidate*. Au moment où j'écris ces lignes, la version finale pour Rails 6.0 n'est pas terminée. J'utilise donc la version la plus récente qui est 6.0.0.0.rc1

Vérifiez que tout fonctionne bien:

```
$ rails -v
Rails 6.0.0.rc1
```

Bases de données

Je vous recommande fortement d'installer Postgresql pour gérer vos bases de données. Mais ici, pour plus de simplicité, nous allons utiliser SQlite. Si vous utilisez Mac OS vous n'avez pas de bibliothèques supplémentaires à installer. Si vous êtes sous Linux, ne vous inquiétez pas, je vous guide:

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

ou

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

[1] La clé GPG vous permet de vérifier l'identité de l'auteur des sources que vous téléchargez.

Initialisation du projet

Vous devez sans doute déjà savoir comment initialiser une application Rails. Si ce n'est pas le cas, jetez un coup d'œil à cette section.

La commande est donc la suivante:

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

> **NOTE** L'option `--api` est apparue lors de la version 5 de Rails. Elle permet de limiter les librairies et *Middleware* inclus dans l'application. Cela permet aussi d'éviter de générer les vues HTML lors de l'utilisation des générateurs de Rails.

Comme vous pouvez le deviner, les commandes ci-dessus généreront les éléments indispensables à votre application Rails. La prochaine étape est d'ajouter quelques gemmes que nous utiliserons pour construire l'API.

Contrôle de version

Rappelez-vous que Git vous aide à suivre et à maintenir l'historique de votre code. Gardez à l'esprit que le code source de l'application est publié sur GitHub. Vous pouvez suivre le projet sur GitHub

Lorsque vous avez utilisé la commande `rails new`, Ruby on Rails a initialisé le répertoire Git pour vous. Cela signifie que vous n'avez pas besoin d'exécuter la commande `git init`.

Il faut néanmoins configurer les informations de l'auteur des *commits*. Si ce n'est pas déjà fait, placez vous dans le répertoire et lancez les commandes suivantes:

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

Rails fournit également un fichier *.gitignore* pour ignorer certains fichiers que nous ne voulons pas suivre. Le fichier *.gitignore* par défaut devrait ressembler à celui illustré ci-dessous :

.gitignore

```
# Ignore bundler config.
/.bundle

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

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

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

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

Une fois Git mis en place, il suffit d'ajouter les fichiers et de valider les modifications. Les commandes nécessaires sont les suivantes:

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

> **NOTE** J'ai appris que commencer un message par un verbe au présent décrit ce que fait le commit et non ce qu'il a fait. De cette façon il est plus facile de lire et de comprendre l'historique du projet (ou du moins pour moi). Je vais suivre cette pratique jusqu'à la fin du tutoriel.

Enfin, et c'est une étape optionnelle, nous déployons le projet sur **GitHub** (je ne vais pas l'expliquer ici) et poussons notre code vers le serveur distant. On commence donc par ajouter un serveur distant:

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

Ensuite nous poussons le code:

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

Au fur et à mesure que nous avançons dans le tutoriel, j'utiliserai les pratiques que j'utilise quotidiennement. Cela inclut le travail avec les branches, le rebasage, le squash et bien d'autres. Vous n'avez pas à vous inquiéter si vous ne connaissez pas tous ces termes, je les expliquerai le temps venu.

Conclusion

Cela a été un chapitre assez long. Si vous êtes arrivés ici, permettez-moi de vous féliciter. Les choses vont s'améliorer à partir de ce point. Commençons à mettre les mains dans le code!

L’API

Dans ce chapitre, je vais vous donner les grandes lignes de l’application. Vous devriez avoir lu le chapitre précédent. Si ce n’est pas le cas, je vous recommande de le faire.

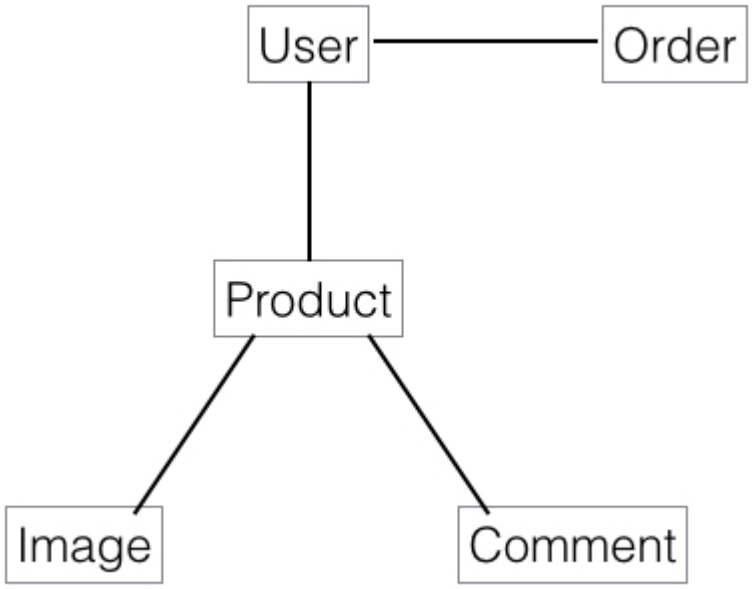
Vous pouvez cloner le projet jusqu’ici avec:

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

Pour résumer, nous avons simplement généré notre application Rails et réalisé notre premier commit.

Planification de l'application

Notre application sera assez simple. Elle se composera de cinq modèles. Ne vous inquiétez pas si vous ne comprenez pas bien ce qui se passe, nous reverrons et développerons chacune de ces ressources au fur et à mesure que nous avancerons avec le tutoriel.



En bref, nous avons l'utilisateur (User) qui sera en mesure de passer de nombreuses commandes (Order), télécharger de multiples produits (product) qui peuvent avoir de nombreuses images (Image) ou commentaires (Comment) d'autres utilisateurs sur l'application.

Nous n'allons pas construire d'interface pour l'interaction avec l'API afin de ne pas surcharger le tutoriel. Si vous voulez construire des vues, il existe de nombreuses options comme des frameworks JavaScript (Angular, Vue.JS, React) ou des librairies mobiles (AFNetworking).

À ce stade, vous devriez vous poser cette question:

> D'accord, mais j'ai besoin d'explorer et de visualiser l'API que je vais construire, non?

C'est juste. Si vous *googlez* quelque chose lié à l'exploration d'une API, vous allez trouver pas mal de résultats. Vous pouvez par exemple utiliser Postman qui est devenu incontournable. Mais nous n'allons pas l'utiliser. Dans notre cas nous allons utiliser **cURL** qui est un outil en ligne de commande disponible presque partout.

Mettre en place l'API

Une API est définie par wikipedia comme *une interface de programmation d'application (API) qui est un ensemble normalisé de composants qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels*. En d'autres termes, il s'agit d'une façon dont les systèmes interagissent les uns avec les autres via une interface (dans notre cas un service web construit avec JSON). Il existe d'autres types de protocoles de communication comme SOAP, mais nous n'en parlons pas ici.

JSON est devenu incontournable en tant que format de fichier pour Internet en raison de sa lisibilité, de son extensibilité et de sa facilité à mettre en œuvre. Beaucoup de frameworks JavaScript l'utilisent comme protocole par défaut comme Angular ou EmberJS. D'autres grandes bibliothèques en Objective-C l'utilisent comme AFNetworking ou RESTKit. Il existe probablement de bonnes solutions pour Android mais en raison de mon manque d'expérience sur cette plate-forme de développement je ne peux pas vous recommander quelque chose.

Nous allons donc utiliser le format JSON pour construire notre API. La première idée qui pourrait vous venir à l'esprit serait de commencer à créer des routes en vrac. Le problème est qu'elles ne seraient pas normalisées. Un utilisateur ne pourrait pas deviner quelle ressource est renvoyée par une route.

C'est pourquoi une norme existe: **REST** *(Representational State Transfer)*. REST impose une norme pour les routes qui créent, lisent, mettent à jour ou suppriment des informations sur un serveur en utilisant de simples appels HTTP. C'est une alternative aux mécanismes plus complexes comme SOAP, CORBA et RPC. Un appel REST est simplement une requête GET HTTP vers le serveur.

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

Et avec REST, vous pouvez appeler une URL avec une requête HTTP spécifique. Dans ce cas avec une requête GET:

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

Les API *RESTful* doivent suivre au minimum trois règles:

- Une URI de base comme http://example.com/resources/

- Un type de média Internet pour représenter les données, il est communément JSON et est communément défini par l'échange d'en-têtes.
- Suivez les méthodes HTTP standard telles que GET, POST, PUT, PUT, DELETE.
  - **GET**: Lit la ou les ressources définies par le modèle URI
  - **POST**: Crée une nouvelle entrée dans la collection de ressources
  - **PUT**: Met à jour une collection ou un membre des ressources
  - **DELETE**: Détruit une collection ou un membre des ressources

Cela peut sembler compliqué mais au fur et à mesure que nous avancerons dans le tutoriel cela deviendra beaucoup plus facile à comprendre.

Routes, contraintes et *Namespaces*

Avant de commencer à taper du code, nous allons préparer le répertoire Git. Le *workflow* que nous allons suivre est le suivant:

- Nous allons créer une branche par chapitre
- Une fois terminé, nous pousserons la branche sur GitHub
- Nous la fusionnerons avec master

Commençons donc par ouvrir le terminal dans le répertoire market_place_api et tapez la commande suivante pour créer la branche:

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

Nous allons seulement travailler sur le fichier config/routes.rb car nous allons simplement définir les contraintes et le format de réponse par défaut pour chaque requête.

config/routes.rb

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

Effacez tout le code commenté qui se trouve dans le fichier. Nous n'en aurons pas besoin. Ensuite, faites un *commit*, juste pour vous échauffer:

```
$ git commit -am "Removes comments from the routes file"
```

Nous allons isoler les contrôleurs API dans des *Namespace*. Avec Rails, c'est assez simple. Il suffit de créer un dossier sous app/controllers nommé api. Le nom est important car c'est le *Namespace* que nous allons utiliser pour gérer les contrôleurs pour les points d'entrée de l'API

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

Nous ajoutons ensuite ce *Namespace* dans notre fichier routes.rb:

config/routes.rb

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

En définissant un *Namespace* dans le fichier routes.rb, Rails mappera automatiquement ce *Namespace* à un répertoire correspondant au nom sous le dossier contrôleur (dans notre cas le répertoire api/).

Les types de medias supportés par Rails

Rails supporte jusqu'à 35 types de médias différents! Vous pouvez les lister en accédant à la classe SET sous le module de Mime:

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

C'est important parce que nous allons travailler avec JSON, l'un des types MIME intégrés par Rails. Ainsi nous avons juste besoin de spécifier ce format comme format par défaut:

config/routes.rb

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

Jusqu'à présent, nous n'avons rien fait de fou. Nous voulons maintenant générer un *base_uri* qui inclut la version de l'API comme ceci: http://localhost:3000/api/v1.

NOTE	Régler l'API sous un sous-domaine est une bonne pratique car cela permet d'adapter l'application à un niveau DNS. Mais dans notre cas, nous allons simplifier les choses pour l'instant.

Vous devriez vous soucier de versionner votre application dès le début

car cela donnera une **meilleure structure** à votre API. Lorsque des changements interviendront sur votre API, vous pouvez ainsi proposer aux développeurs de s'adapter aux nouvelles fonctionnalités pendant que les anciennes sont dépréciées.

config/routes.rb

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

Les conventions des API

Vous pouvez trouver de nombreuses approches pour configurer la `base_uri` d'une API. En supposant que nous versionnons notre api:

- `api.example.com/`: Je suis d'avis que c'est la voie à suivre, elle vous donne une meilleure interface et l'isolement, et à long terme peut vous aider à mettre rapidement à l'échelle
- `example.com/api/`: Ce modèle est très commun. C'est un bon moyen de commencer quand vous ne voulez pas de *Namespace* de votre API avec sous un sous-domaine
- `example.com/api/v1`: Cela semble être une bonne idée. Définir la version de l'API par l'URL semble être un modèle plus descriptif. Cependant, vous forcez à inclure la version à l'URL sur chaque demande. Cela devient un problème si vous décidez de changer ce modèle

Il est temps de faire un *commit*:

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

Afin de définir la version de l'API, nous devons d'abord ajouter un autre répertoire sous le dossier `api/` que nous avons créé:

```
$ mkdir app/controllers/api/v1
```

L'API est désormais *scopée* via l'URL. Par exemple, avec la configuration actuelle, la récupération d'un produit via l'API se ferait avec cette url: http://localhost:3000/v1/products/1.

Ne vous inquiétez pas, nous rentrerons plus en détails à propos du versionnement plus tard. Il est temps de *commiter*:

```
$ git commit -am "Set the routes namespaces for the api"
```

> **NOTE**
>
> Il existe certaines pratiques dans la construction d'API qui recommandent de ne pas versionner l'API via l'URL. C'est vrai. Le développeur ne devrait pas être au courant de la version qu'il utilise. Dans un souci de simplicité, j'ai choisi de mettre de côté cette convention que nous pourrons appliquer dans un second temps.

Nous arrivons à la fin de notre chapitre. Il est donc temps d'appliquer toutes nos modifications sur la branche master en faisant un *merge*. Pour cela, on se place sur la branche master et on *merge* chapter02:

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

Conclusion

Ça a été un peu long, je sais, mais vous avez réussi! N'abandonnez pas, c'est juste notre petite fondation pour quelque chose de grand, alors continuez comme ça. Sachez qu'il y a des gemmes qui gèrent ce genre de configuration pour nous:

- RocketPants
- Versionist

Je n'en parle pas ici puisque nous essayons d'apprendre comment mettre en œuvre ce genre de fonctionnalité.