

Initiation à Rails

par Yann Marec ([Yann Marec](#))

Date de publication : 16/03/2006

Dernière mise à jour :

Rails est un framework MVC basé sur Ruby. Cet article va nous permettre de le découvrir à travers le développement d'une petite application.

- I - Présentation
- II - installer Rails
- III - Configuration de l'application
- IV - Validation de formulaire
- V - Modification des vues
- VI - Mapping des relations 1-n
- VII - Conclusion
- VIII - Remerciements

I - Présentation

Ruby on Rails, ou RoR ou Rails, est un framework web basé sur le design pattern **MVC** et utilisant le langage **Ruby**.

Ruby est un langage de programmation interprété orienté objet.

Toute donnée est un objet, y compris les types primitifs.

Rails a été conçu avec l'idée de respecter deux principes: **DRY** ou "Don't Repeat Yourself" (Ne pas se répéter) et « Convention plutôt que Configuration ».

Le premier prône la réutilisation du code déjà existant.

Le deuxième force les développeurs à respecter des conventions de nommage. C'est à première vue une contrainte, mais cela va permettre à Rails de prendre en charge la configuration de l'application au lieu de laisser cette tâche aux développeurs.

Ces conventions doivent être suivies mais peuvent être contournées au prix de plus de configuration dans les fichiers.

Rails, par exemple, suppose que les noms des tables sont au pluriel et écrits en minuscule.

Cette convention permet à Rails de faire directement un mapping entre les tables et les classes, sans obliger le développeur à décrire ce mapping dans un fichier xml comme cela doit être fait avec Hibernate par exemple.

Le framework MVC Rails est complet et propose des outils pour chacune des couches de l'application.

Le modèle est géré par le composant **ActiveRecord**, le contrôleur et la vue par **Action Pack**, les web services par le composant **Action Web Service**. **Ajax** est intégré avec le composant **Prototype**.

Rails est aussi livré avec des générateurs de code, très pratiques pour créer rapidement les bases de notre application.

II - installer Rails

On peut télécharger l'installation tout-en-un de Ruby sur le site <http://rubyinstaller.rubyforge.org/wiki/wiki.pl> .

Il suffit d'installer Ruby en suivant les recommandations de l'installateur.

Pour tester la bonne installation de Ruby, dans une invite de commandes, tapez la commande "**irb**";

irb est un interpréteur de commandes ruby, comme il en existe pour le langage Caml ou des langages fonctionnels comme Scheme ou Lisp.

Passons à l'installation de Rails. La procédure d'installation est la même que pour les autres packages de Ruby.

Dans une invite de commandes, tapez la commande

```
gem install rails --include-dependencies.
```

gem téléchargera et installera lui même le package rails ainsi que tous les packages dont dépend Rails.

Tout est prêt pour mettre en place notre première application Rails.

III - Configuration de l'application

L'application est un annuaire gérant la liste des collaborateurs d'une entreprise.

Pour le stockage des données, nous allons créer une base de données Mysql annuaire.

Dans cette base de données, nous créons une table **collaborateurs**. (le pluriel est important pour le nom de la base, Rails imposant certaines règles de nommage)

Voici le code SQL permettant la création de la table :

```
CREATE TABLE collaborateurs (
  id int(11) NOT NULL auto_increment,
  nom varchar(255) NOT NULL,
  prenom varchar(255) NOT NULL,
  dateNaissance date default NULL,
  telephone varchar(255) default NULL,
  mail varchar(255) default NULL,
  PRIMARY KEY (id)
)
```

	Datatype	Len	Default	PK?	Binary?	Not Null?	Unsigned?	Au
	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
	date			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
	varchar	255		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
				<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Table Collaborateurs

Rails impose d'avoir un champ **id** comme clé primaire (primary key ou PK) auto-incrémentée.

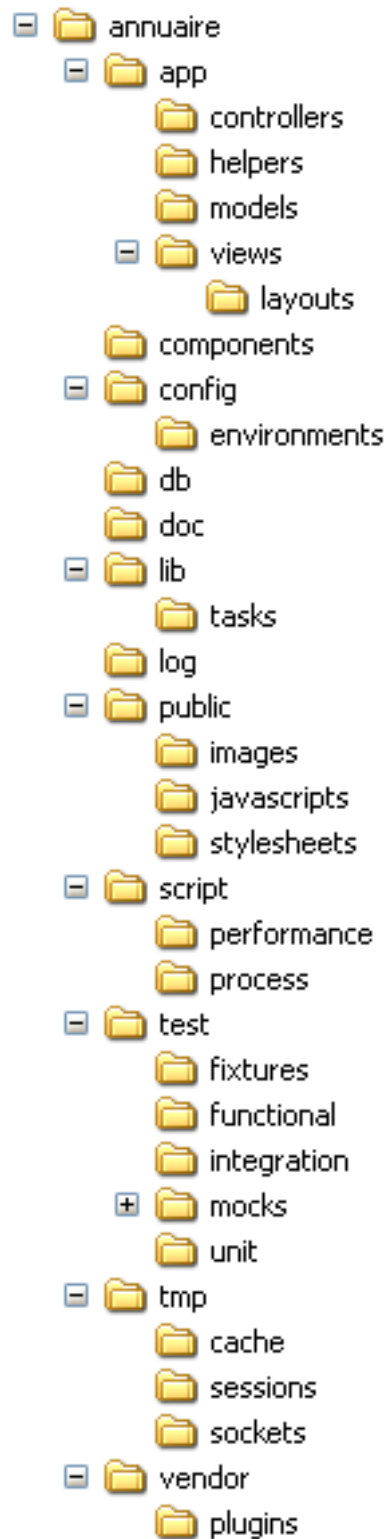
Ensuite, dans une console, placez-vous dans le même répertoire que l'application.

Tapez la commande :

rails annuaire

Rails va créer pour nous le squelette de notre application.

Allez dans le répertoire « annuaire » nouvellement créé.



Répertoire de l'application

Dans le répertoire **app** se trouveront nos fichiers ruby.

Rails va nous aider à les créer, il va notamment générer pour nous les fichiers de mapping avec la base.

Dans le fichier config/database.yml se trouvent les informations pour se connecter à la base.

Par défaut, rails configure l'accès à 3 bases, une pour le développement, une pour la production et une pour les tests.

Seule celle pour le développement nous intéresse pour le moment.

Au besoin, remaniez les informations comme username, password, host et database.

Dans notre cas, le fichier database.yml est modifié. Le nom de la base est annuaire et non annuaire_development.

Pour créer ensuite un **CRUD** (Create, Read, Update, Delete) permettant de manipuler les données de notre base avec les opérations "create", "read", "update" et "delete", dans une console, allez dans le répertoire annuaire et tapez la commande :

```
ruby script\generate scaffold Collaborateur
```

Le script ruby **generate** va générer pour nous le modèle, le contrôleur et les vues de notre **CRUD**.

Il ne reste alors plus qu'à le tester.

Lançons pour ça le serveur d'applications que Rails utilise (**Webrick**) :

```
ruby script\server
```

Dans votre navigateur web, allez à l'URL :

```
http://localhost:3000/collaborateurs
```

Vous avez déjà un **CRUD** basique qui marche sur le serveur d'applications **WebRick**, idéal pour le développement.

Lorsque vous souhaiterez passer en production, il sera possible de passer sur un serveur **Apache** par exemple.

Le système des URL de l'application est très simple à manipuler.

Après l'URL de l'application (en développement, **http://localhost:3000**), on ajoute le nom du contrôleur et ensuite le nom de la méthode du contrôleur.

Par exemple, pour créer un nouveau collaborateur, nous irons à l'URL **http://localhost:3000/collaborateurs/new**.

Pour avoir la fiche du collaborateur ayant l'id 1, on va utiliser la méthode show et lui donner comme paramètre 1 donc **http://localhost:3000/collaborateurs/show/1**

IV - Validation de formulaire

Pour rester cohérent avec la base de données, nous devons exiger que les champs du formulaire Nom et Prenom de la page edit ou new ne soient pas nuls.

Les classes de la partie **Modèle** du **MVC** se trouvent dans le répertoire annuaire/app/models.

On aura une classe ruby pour chaque table SQL.

Nous avons donc une classe Collaborateur.

Modifions le fichier app/models/collaborateur.rb:

```
class Collaborateur < ActiveRecord::Base
  validates_presence_of :nom, :prenom
end
```

La méthode **validates_presence_of** est fournie par Rails et permet de vérifier que les champs passés en arguments soient bien présents lors de la validation du formulaire.

Il existe plusieurs méthodes permettant de vérifier plusieurs choses lors de la validation du formulaire, comme par exemple le format d'un champ (nous nous en servons par la suite), la longueur d'un champ, etc..

Une liste de ces méthodes est disponible dans la documentation de Rails à cette adresse : <http://api.rubyonrails.org/classes/ActiveRecord/Validations/ClassMethods.html#M000941>

Maintenant, si le formulaire est validé avec un des champs nom ou prenom vide, nous aurons le message d'erreur suivant :

New collaborateur

2 errors prohibited this collaborateur from being saved

There were problems with the following fields:

- Nom can't be blank
- Prenom can't be blank

Nom

Prenom

Datenaissance

Telephone


Mail

Create

[Back](#)

Erreur de validation

Nous souhaiterions un contrôle sur le champ mail pour qu'il respecte le format des adresses mails.

Nous allons donc utiliser cette fois-ci la méthode de validation **validates_format_of**, qui prend en argument le champ à vérifier et une  **expression régulière** à tester sur ce champ. Modifions le fichier app/models/collaborateur.rb:

```
class Collaborateur < ActiveRecord::Base
  validates_presence_of :nom, :prenom
  validates_format_of :mail, :with => /^[^@\\s]+@((?:[-a-z0-9]+\\.)+[a-z]{2,})$/
end
```

On aura alors ce message d'erreur si jamais le format du mail est invalide :

New collaborateur

1 error prohibited this collaborateur from being saved

There were problems with the following fields:

- Mail is invalid

Nom

Prenom

Datenaissance

Telephone

Mail

[Back](#)

Erreur de validation du mail

V - Modification des vues

On veut maintenant modifier l'affichage d'une des pages, par exemple celle listant les collaborateurs.

Rails nous a généré les fichiers .rhtml qui sont des pages html agrémentées de code ruby dans le répertoire app/views (partie Vue du MVC).

Modifions donc le fichier app/views/collaborateur/list.rhtml pour que la liste n'affiche que le nom et le prénom du collaborateur et que ce nom+prénom soit un lien vers la fiche complète du collaborateur:

```
<h1>Listing collaborateurs</h1>
<table>
<% for collaborateur in @collaborateurs %>
  <tr>
  <td>
    <%= link_to collaborateur.nom + ' ' + collaborateur.prenom ,
      :action => 'show',
      :id => collaborateur %>
  </td>
  <td>
    <%= link_to 'Edit',
      :action => 'edit',
      :id => collaborateur %>
  </td>
  <td>
    <%= link_to 'Destroy',
      { :action => 'destroy', :id => collaborateur },
      :confirm => 'Are you sure?',
      :post => true %>
  </td>
</tr>
</table>

<%= link_to 'Previous page',
  { :page => @collaborateur_pages.current.previous } if @collaborateur_pages.current.previous %>
<%= link_to 'Next page',
  { :page => @collaborateur_pages.current.next } if @collaborateur_pages.current.next %>

<br />

<%= link_to 'New collaborateur', :action => 'new' %>
```

Pour que le champ date de notre formulaire soit plus « francophone », il faut modifier le fichier app/views/collaborateur/_form.rhtml et mettre plus d'options sur le date_select :

```
<%= date_select
  'collaborateur',
  'dateNaissance',
  :order => [:day, :month, :year],
  :use_month_numbers => true,
  :start_year => 1900,
  :end_year => Date.today.year,
  :include_blank => true%>
```

Cette instruction génère un champ **SELECT** dans notre formulaire html, les arguments nous permettent de le configurer à notre convenance.

le **date_select** va chercher l'information dateNaissance dans la table collaborateur.

L'ordre des select sera (jour, mois, année), les noms des mois seront remplacés par leurs numéros,

la fourchette des années sera [1900;année en cours] et l'utilisateur a la possibilité de ne pas renseigner l'un des select.

Pour que le format d'affichage des dates soit par défaut jj/mm/yyyy et que les noms des jours et mois soient en français, il faut ajouter à la fin du fichier config/environment.rb cette ligne :

```
require 'overrides'
```

et ajouter le fichier lib/overrides.rb :

```
Date::MONTHS = {
  'Janvier' => 1,
  'Fevrier' => 2,
  'Mars' => 3,
  'Avril' => 4,
  'Mai' => 5,
  'Juin' => 6,
  'Juillet' => 7,
  'Aout' => 8,
  'Septembre'=> 9,
  'Octobre' =>10,
  'Novembre' =>11,
  'Decembre' =>12 }
Date::DAYS = {
  'Lundi' => 0,
  'Mardi' => 1,
  'Mercredi' => 2,
  'Jeudi'=> 3,
  'Vendredi' => 4,
  'Samedi' => 5,
  'Dimanche' => 6 }
Date::ABBR_MONTHS = {
  'jan' => 1,
  'fev' => 2,
  'mar' => 3,
  'avr' => 4,
  'mai' => 5,
  'juin' => 6,
  'juil' => 7,
  'aou' => 8,
  'sep' => 9,
  'oct' =>10,
  'nov' =>11,
  'dec' =>12 }
Date::ABBR_DAYS = {
  'lun' => 0,
  'mar' => 1,
  'mer' => 2,
  'jeu' => 3,
  'ven' => 4,
  'sam' => 5,
  'dim' => 6 }
Date::MONTHNAMES = [nil] + %w(Janvier Fevrier Mars Avril Mai Juin Juillet Aout Septembre Octobre
  Novembre Decembre )
Date::DAYNAMES = %w(Lundi Mardi Mercredi Jeudi Vendredi Samedi Dimanche )
Date::ABBR_MONTHNAMES = [nil] + %w(jan fev mar avr mai juin juil aou sep oct nov dec)
Date::ABBR_DAYNAMES = %w(lun mar mer jeu ven sam dim)

class Time
  alias :strftime_nolocale :strftime
```

```
def strftime(format)
  format = format.dup
  format.gsub!(/%a/, Date::ABBR_DAYNAMES[self.wday])
  format.gsub!(/%A/, Date::DAYNAMES[self.wday])
  format.gsub!(/%b/, Date::ABBR_MONTHNAMES[self.mon])
  format.gsub!(/%B/, Date::MONTHNAMES[self.mon])
  self.strftime_nolocale(format)
end
end
```

VI - Mapping des relations 1-n

Nous allons ajouter une table fonctions à notre base de données pour illustrer les relations 1-n.

Un collaborateur exerce une fonction et une fonction peut être exercée par plusieurs collaborateurs.

```
CREATE TABLE fonctions (
  id int(11) NOT NULL auto_increment,
  titre varchar(255) NOT NULL,
  PRIMARY KEY (id)
)
```

datatype	Len	Default	PK?	Binary?	Not Null?	Unsigned?	Auto Incr?	Z
int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
varchar	255		<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Comme pour collaborateurs, nous devons avoir une colonne id comme clé primaire et auto-incrémentée.

De plus, il faut rajouter une colonne **fonction_id** à la table collaborateurs, qui sera une clé étrangère pointant vers un enregistrement de la table fonctions.

Dans une console, nous lançons la commande :

ruby script\generate scaffold fonction

Un **CRUD** est alors disponible pour les fonctions à l'URL : **http://localhost:3000/fonctions**

Vous pouvez l'utiliser pour y entrer des fonctions qui seront ensuite affectées aux collaborateurs.

Pour faire la jointure entre les deux classes modèles, il faut ajouter une ligne dans chacune d'entre elles :

Dans le fichier app/models/collaborateur.rb :

```
class Collaborateur < ActiveRecord::Base
  belongs_to :fonction

  validates_presence_of :nom, :prenom
  validates_format_of :mail, :with => /^[^@\s]+@((?:[-a-z0-9]+\.)+[a-z]{2,})$/
end
```

La méthode **belongs_to** indique alors qu'un collaborateur "appartient" à une fonction, c'est à dire que la clé étrangère fonction de la classe Collaborateur appartient à la table Fonctions.

Dans le fichier app/models/fonction.rb :

```
class Fonction < ActiveRecord::Base
  has_many :collaborateurs
end
```

Cette fois-ci, nous utilisons la méthode **has_many**, car une fonction peut "posséder" plusieurs collaborateurs, c'est à dire qu'à une fonction, on fait correspondre plusieurs collaborateurs.

Il faut maintenant modifier le contrôleur de collaborateur pour qu'il prenne en compte les fonctions.

Les classes contrôleurs (une par table SQL) se trouvent dans le répertoire app/controllers.

On modifie donc le fichier app/controllers/collaborateurs_controller.rb et en particulier sa méthode **edit** :

```
def edit
  @collaborateur = Collaborateur.find(params[:id])
  @fonctions = Fonction.find_all
end
```

Fonction.find_all est une méthode de la classe modèle Fonction qui renvoie la liste de toutes les fonctions.

La vue utilisera cet attribut @fonctions pour remplir le select du formulaire.

Logiquement, il faut donc faire de même dans la méthode **new** :

```
def new
  @collaborateur = Collaborateur.new
  @fonctions = Fonction.find_all
end
```

Il faut finalement ajouter un select au fichier app/views/collaborateurs/_form.rhtml :

```
<p>
<b>Fonction:</b><br>
<select name="collaborateur[fonction_id]">
  <% @fonctions.each do |fonction| %>
    <option value="<%= fonction.id %>"
      <%= ' selected' if fonction.id == @collaborateur.fonction_id %>>
      <%= fonction.titre %>
    </option>
  <% end %>
</select>
</p>
```

et afficher cette nouvelle information dans la fiche d'un collaborateur dans le fichier app/views/collaborateurs/show.rhtml :

```
<p><b>Fonction: </b><%= @collaborateur.fonction.titre %></p>
```

VII - Conclusion

Nous avons vu que Rails était très productif, car nous avons eu une application utilisable très rapidement.

Nous avons vu que l'application générée était très simple à modifier pour l'adapter à nos besoins. L'architecture de l'application respecte le modèle MVC, très important pour que l'application soit facilement maintenable.

Rails peut être utilisé pour réaliser rapidement des maquettes d'applications mais le framework a aussi montré qu'il était utilisable pour développer de grosses applications (exemples : Basecamp, Odeo, List Apart, Scoopeo.com, etc..)

VIII - Remerciements

Je tiens à remercier **titoumimi** pour ses conseils qu'il m'a apporté tout au long de la rédaction de cet article ainsi que **jc_cornic** et **fearyourself** pour leurs relectures.

