13 fév 2009

Introduction à Ruby on Rails

Catégorie : Programmation Tags : lmhs

~~Retrouvez cet article dans : Linux Magazine Hors série 33~~

Faire un article sur Ruby on Rails, alors que la version 2.0 est sur le point de sortir est un challenge un peu risqué ! Dans cet article, nous utiliserons la version 1.2.3 de ce framework. Il ne s'agira que d'une introduction. En effet, Ruby on Rails est très riche et la place est limitée. Cependant, les concepts que nous allons voir forment une bonne base pour comprendre comment s'articule une application développée avec Rails.

1. Présentation de Ruby on Rails

Si nous regardons comment ont été développés et mis au point la majorité des frameworks, nous remarquons souvent qu'il s'agit d'un développement qui n'est, à la base, pas centré sur un besoin spécifique, mais sur l'envie d'un ou plusieurs développeurs de fournir une nouvelle solution à des problématiques génériques. Ce n'est pas le cas de Ruby on Rails.
À l'origine, l'idée n'était pas de fournir un framework, mais de créer un site de gestion de projets : Basecamp [1]. Ce développement a été réalisé par David Heinemeier Hansson pour la société 37signals. Ce n'est que plus tard qu'il a été décidé d'extraire de Basecamp les principes mis au point pour son développement et de les rendre indépendants dans un framework. Ce travail a abouti à la publication de la première release publique, sous licence libre, de Ruby on Rails en juillet 2004. Il aura par la suite fallu un an et demi de travail pour mettre au point la version 1.0 poussé par une communauté grandissante et très rapidement convaincue par la puissance du framework. Nous sommes aujourd'hui à la version 1.2.4 qui préfigure la version 2.0.
Ruby on Rails est donc né avec un passif basé sur des besoins réels et non en extrapolant les besoins auxquels il pourrait répondre et les utilisations qui en découleraient. Si certains pouvaient alors penser que Ruby on Rails a été développé pour répondre à des besoins spécifiques, il s'avère en fait que le framework est suffisamment bien pensé pour pouvoir répondre à tous les besoins d'une application Web moderne, voire bien plus...
Bien plus, parce qu'il est constitué d'un ensemble de composants que le découpage rend utilisables bien au-delà du simple développement Web, et ayant chacun un rôle bien précis. Il y a six grands composants dans Rails.
Les deux premiers ne sont absolument pas spécifiques au développement Web :

- ~~ActiveRecord~~. Ce module regroupe tout un ensemble de classes permettant de faire de l'ORM (Object Relational Mapping). Ainsi, avec ~~ActiveRecord~~, vous pouvez oublier le SQL et ne penser à vos données qu'en qualité d'objet. L'ORM n'est pas l'apanage des développements Web, et, vous pouvez, sans rougir, utiliser ~~ActiveRecord~~ en dehors de Rails. RubyCocoa, par exemple, utilise ce module depuis peu afin de permettre aux développeurs Mac de mimer une utilisation de type CoreData... Passons ;)

- ~~ActiveSupport~~ propose, quant à lui, tout un ensemble d'extensions pour la bibliothèque standard du langage. Nous y trouverons des extensions pour la manipulation des tableaux, des hachages, des entiers, des chaînes de caractères... Là encore, vous pouvez en faire bénéficier tous vos développements sans restriction. Par exemple, si vous souhaitez savoir quelle heure il était il y a 16 minutes, vous pouvez faire ceci :

```
require 'active_support'

16.minutes.ago
# => Fri Oct 05 23:36:00 0200 2007

puts Time::now
# => Fri Oct 05 23:52:15 +0200 2007
```

Les trois autres modules sont eux intimement liés à Rails et sont en fait constitués d'un ensemble de classes servant à simplifier l'écriture de code dans une application Rails.

- ~~ActionView~~ propose tout un ensemble de helpers. Nous explorerons certains d'entre eux dans cet article. Sachez que si vous voulez faire de l'AJAX, c'est dans la documentation d'~~ActionView~~ qu'il faudra piocher !
- ~~ActionController~~ nous simplifiera énormément la vie quand nous créerons nos contrôleurs.
- ~~ActionMailer~~ permet de faciliter l'envoi et la réception de mails.
- ~~ActionWebService~~ nous permet de créer des WebServices de façon ultra simple.

Nous survolerons seulement les composants essentiels pour développer une application Rails dans cet article (et donc, malheureusement pas ~~ActionMailer~~ et ~~ActionWebService~~). Mais cela devrait être suffisant pour vous faire comprendre à quel point la

vie est simple avec Rails !

2. Installation

Tout au long de cet article, nous allons nous amuser à développer une application de gestion de contacts. Nous allons, en fait, nous limiter, dans un premier temps, à créer un " catalogue " de personnes avec, pour chacune, son nom, son prénom, son adresse mail et son numéro de téléphone. Dans sa première version, notre application sera relativement spartiate, ce qui ne veut pas dire qu'elle ne devra pas rendre les services minimums que sont l'ajout, la modification et la suppression de contacts, plus un moyen d'en afficher la liste complète.

Avant de commencer, il faut bien entendu installer Rails. Pour cela, je vous conseille d'utiliser RubyGems en précisant que vous souhaitez installer les dépendances :

```
$ sudo gem install rails -y
Bulk updating Gem source index for: http://gems.rubyforge.org
Successfully installed rails-1.2.4
Successfully installed activesupport-1.4.3
Successfully installed activerecord-1.15.4
Successfully installed actionpack-1.13.4
Successfully installed actionmailer-1.3.4
Successfully installed actionwebservice-1.2.4
Installing ri documentation for activesupport-1.4.3...
Installing ri documentation for activerecord-1.15.4...
Installing ri documentation for actionpack-1.13.4...
Installing ri documentation for actionmailer-1.3.4...
Installing ri documentation for actionwebservice-1.2.4...
Installing RDoc documentation for activesupport-1.4.3...
Installing RDoc documentation for activerecord-1.15.4...
Installing RDoc documentation for actionpack-1.13.4...
Installing RDoc documentation for actionmailer-1.3.4...
Installing RDoc documentation for actionwebservice-1.2.4...
$
```

Comme vous pouvez le voir, l'installation de rails a entraîné l'installation des packages ~~activesupport~~, ~~activerecord~~, ~~actionpack~~, ~~actionmailer~~ et ~~actionwebservices~~. Pour ceux qui se demandent où sont passés ~~ActionView~~ et ~~ActionController~~, ils sont dans ~~actionpack~~.

Rails étant installé, nous pouvons mettre en place le squelette de notre application. Il suffit d'invoquer la commande ~~rails~~ en lui passant en argument le nom de notre application :

```
$ rails address_book
      create
      create  app/controllers
      create  app/helpers
      create  app/models
      create  app/views/layouts
      create  config/environments
      ...
      create  log/server.log
      create  log/production.log
      create  log/development.log
      create  log/test.log
$
```

Nous reviendrons un peu plus tard sur les fichiers générés et leur organisation. Pour le moment, nous allons nous concentrer sur l'essentiel.

3. Base de données et migration

La première chose que nous devons faire maintenant consiste à créer le schéma de notre base de données. Ceci implique de savoir quelle base nous voulons utiliser. Rails est très ouvert sur le sujet, et vous avez, en standard, le choix entre MySQL, PostgreSQL, Oracle ou SQLite (2 ou 3). Mais si vous préférez SQL Server (sic), Informix, DB2, Firebird, Sybase ou toute autre base, il faudra installer l'adaptateur correspondant. Vous trouverez toutes les informations nécessaires sur le site [2] de Ruby on Rails. Dans cette présentation, nous ferons le choix d'SQLite3. Cette base est minimaliste, mais couvre largement nos besoins.

Le paramétrage de la base de données se fait au moyen du fichier ~~config/database.yml~~ situé dans le répertoire de notre application. Plaçons-nous dans le répertoire ~~address_book~~ et éditons ce fichier :

```
# MySQL (default setup).  Versions 4.1 and 5.0 are recommended.
#
development:
  adapter: mysql
  database: address_book_development
  username: root
  password:
  host: localhost

# Warning: The database defined as 'test' will be erased and
# re-generated from your development database when you run 'rake'.
# Do not set this db to the same as development or production.
test:
  adapter: mysql
```

```
  database: address_book_test
  username: root
  password:
  host: localhost

production:
  adapter: mysql
  database: address_book_production
  username: root
  password:
  host: localhost
```

Comme vous pouvez le voir, ce fichier est en YAML[3]. Je vous laisse consulter la documentation de ce langage. Mais ne perdez pas trop de temps, car c'est la seule fois que nous allons voir un tel fichier et il est suffisamment lisible pour ne pas nous imposer de devenir un spécialiste du domaine.

Rails nous a préparé une configuration pour MySQL. C'est en effet la base "par défaut". Nous allons donc modifier le fichier en précisant que nous utilisons l'adaptateur sqlite3. Nous allons également indiquer quels sont les fichiers de base de données. Voici à quoi doit ressembler ce fichier après modification.

```
development:
  adapter: sqlite3
  database: db/address_book_development.sqlite3

test:
  adapter: sqlite3
  database: db/address_book_test.sqlite3

production:
  adapter: sqlite3
  database: db/address_book_production.sqlite3
```

Sachez que nous aurions très bien pu demander à Rails de créer directement un fichier de paramétrage pour sqlite3. Il suffit pour cela d'utiliser l'option ~~--database=~~ (ou ~~-d~~) de la commande ~~rails~~, suivie du type de base que nous souhaitons utiliser. Dans notre cas, cela aurait donné :

```
       rails address_book -d sqlite3
```

Nous aurions ainsi économisé l'édition et la modification de ce fichier. Parfois, chaque seconde compte ;)

Maintenant que nous avons précisé quelle base nous voulons utiliser, nous pouvons créer le schéma. Nous avons besoin pour le moment d'une seule table dans laquelle nous allons stocker nos contacts.

Habituellement, pour créer un schéma de base de données, nous devons avoir les notions de SQL nécessaires à l'écriture d'un script de création de nos tables. Avec Ruby on Rails, nous allons nous contenter de code Ruby.

De plus, comme je l'ai signalé un peu plus haut, Rails utilise les principes de l'ORM. Il nous faut donc également un objet faisant le mapping avec notre table. Pour créer ces différents éléments, nous allons utiliser la commande ~~generate~~ de rails en lui précisant que nous voulons un modèle contact :

```
$ ruby script/generate model contact
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/contact.rb
      create  test/unit/contact_test.rb
      create  test/fixtures/contacts.yml
      create  db/migrate
      create  db/migrate/001_create_contacts.rb
$
```

Le seul fichier que nous allons regarder pour le moment est ~~db/migrate/001_create_contacts.rb~~. C'est lui qui décrit le schéma SQL de notre table, mais, comme promis, en utilisant une syntaxe purement Ruby :

```
class CreateContacts < ActiveRecord::Migration
  def self.up
    create_table :contacts do |t|
    end
  end

  def self.down
    drop_table :contacts
  end
end
```

Il n'y a pour le moment pas grand-chose dans ce fichier. Il contient en fait une déclaration de classe (~~CreateContacts~~) comprenant elle-même deux méthodes : ~~up~~ et ~~down~~. Nous remarquons immédiatement que nous demandons, dans la méthode ~~up~~, de créer une table contacts (~~:contacts~~). C'est le sens de l'appel à la méthode ~~create_table~~. Notez bien la présence du s à la fin de ~~:contacts~~. En effet, nous avons demandé à Rails de créer un modèle contact (sans ~~s~~ donc) et pourtant rails a pluralisé le nom de notre table. C'est tout à fait logique en fait, car nous allons bien y stocker plusieurs enregistrements ! Ne modifions rien pour le moment et retournons dans notre console pour exécuter la commande suivante :

```
$ rake db:migrate
(in /Users/greg/Desktop/address_book)
== CreateContacts: migrating ==================================
-- create_table(:contacts)
```

```
   -> 0.0871s
== CreateContacts: migrated (0.0872s) =======================
$
```

Voilà, nous venons de créer notre table contacts dans la base SQL. Vous pouvez le vérifier :

```
$ sqlite3 db/address_book_development.sqlite3 <<< .dump
BEGIN TRANSACTION;
CREATE TABLE schema_info (version integer);
INSERT INTO "schema_info" VALUES(1);
CREATE TABLE contacts ("id" INTEGER PRIMARY KEY NOT NULL);
COMMIT;
$
```

Conformément à ce que nous avons, ou plutôt ce que nous n'avons pas, écrit dans le fichier ~~db/migrate/001_create_contacts.rb~~, nous nous retrouvons avec une table ~~contacts~~. Bien que nous n'ayons rien demandé, cette table possède un champ ~~id~~. Ce champ est automatiquement ajouté par Rails. Il constitue la clé primaire de notre table.

Il est maintenant temps d'ajouter nos champs nom, prénom, email et téléphone. Pour cela, nous pourrions être tenté de modifier le fichier ~~001_create_contacts.rb~~. Eh bien soit, allons-y...

```
class CreateContacts < ActiveRecord::Migration
  def self.up
    create_table :contacts do |t|
      t.column :nom,       :string
      t.column :prenom,    :string
      t.column :email,     :string
      t.column :telephone, :string
    end
  end

  def self.down
    drop_table :contacts
  end
end
```

L'ajout des colonnes se fait en utilisant la méthode ~~column~~ prenant en paramètre le nom et le type de chaque colonne. En plus de ces deux paramètres obligatoires, vous pouvez préciser différentes options permettant de définir si le champ peut être nul ou non, sa taille... Je vous laisse regarder la documentation [4] des API Rails pour plus de détails. Dans notre table, nous n'utilisons que des champs de type chaîne de caractères (~~:string~~). Bien entendu, vous pouvez en utiliser d'autres : ~~:primary_key~~, ~~:string~~, ~~:text~~, ~~:integer~~, ~~:float~~, ~~:decimal~~, ~~:datetime~~, ~~:timestamp~~, ~~:time~~, ~~:date~~, ~~:binary~~, ~~:boolean~~. Encore une fois, je vous renvoie vers la documentation...

Relançons maintenant la commande ~~rake db:migrate~~ et regardons ce qui s'est passé :

```
$ rake db:migrate
(in /Users/greg/Desktop/address_book)
$ sqlite3 db/address_book_development.sqlite3 <<< .dump
BEGIN TRANSACTION;
CREATE TABLE schema_info (version integer);
INSERT INTO "schema_info" VALUES(1);
CREATE TABLE contacts ("id" INTEGER PRIMARY KEY NOT NULL);
COMMIT;
$
```

Rien ! Pourquoi ?

Simplement parce que nous avions déjà une table contacts. En effet, Rails utilise un principe de migration qui permet de "versionner" le schéma de la base de données. C'est le sens de la table ~~schema_info~~ que nous avons ignorée jusqu'à maintenant. Cette table contient un seul champ (~~version~~) dans lequel est indiqué le numéro de version de notre schéma. Or, nous souhaitons le faire évoluer. Nous sommes en version 1, il faut passer en version 2. Pour faire cela, nous allons donc créer une nouvelle version de notre schéma. Là encore, nous allons utiliser le script ~~generate~~ en lui demandant de créer un fichier de migration :

```
$ ruby script/generate migration contact_informations
      exists  db/migrate
      create  db/migrate/002_contact_informations.rb
$
```

Le paramètre passé après ~~migration~~ est purement décoratif et ne sert qu'à décrire ce que nous voulons faire. Dans ce cas présent, nous voulons ajouter les informations pour nos contacts.

Nous avons un nouveau fichier : ~~db/migrate/002_contact_informations.rb~~. C'est dans ce fichier que nous allons demander l'ajout de nos colonnes dans la table ~~contacts~~. Ouvrez votre éditeur et modifiez ce fichier de la façon suivante :

```
class ContactInformations < ActiveRecord::Migration
  def self.up
    create_table :contacts do |t|
      t.column :nom,        :string
      t.column :prenom,     :string
      t.column :email,      :string
      t.column :telephone,  :string
    end
  end

         def self.down
  end
end
```

Relançons la demande de migration :

```
$ rake db:migrate
(in /Users/greg/Desktop/address_book)
== ContactInformations: migrating ============================================
-- create_table(:contacts)
rake aborted!
SQLite3::SQLException: table contacts already exists: CREATE TABLE contacts ("id" INTEGER PRIMARY KEY NOT NULL, "nom" varchar(255) DEFAULT NULL, "prenom" varchar(255) DI

(See full trace by running task with —trace)
$
```

Ca plante ! Et c'est tout à fait normal, car la table ~~contacts~~ existe déjà. En fait, il faut forcer la demande. Pour cela, nous devons ajouter une option à la méthode ~~create_table~~, indiquant qu'il faut forcer la création de la table :

```
class ContactInformations < ActiveRecord::Migration
  def self.up
    create_table :contacts, :force => true do |t|
      t.column :nom,        :string
      t.column :prenom,     :string
      t.column :email,      :string
      t.column :telephone,  :string
    end
  end

  def self.down
  end
end
```

Prenez la bonne habitude d'ajouter systématiquement cette option. D'ailleurs, modifiez tout de suite le fichier ~~db/migrate/001_create_contacts.rb~~.

Cette fois-ci, cela fonctionne parfaitement, et nous passons bien en version 2 de notre schéma :

```
$ rake db:migrate
(in /Users/greg/Desktop/address_book)
== ContactInformations: migrating ============================================
-- create_table(:contacts, {:force=>true})
   -> 0.5288s
== ContactInformations: migrated (0.5289s) ===================================

$ sqlite3 db/address_book_development.sqlite3 <<< .dump
BEGIN TRANSACTION;
CREATE TABLE schema_info (version integer);
INSERT INTO "schema_info" VALUES(2);
CREATE TABLE contacts ("id" INTEGER PRIMARY KEY NOT NULL, "nom" varchar(255) DEFAULT NULL, "prenom" varchar(255) DEFAULT NULL, "email" varchar(255) DEFAULT NULL, "telepl
COMMIT;
$
```

Vous avez certainement compris que les noms des fichiers contenus dans le répertoire ~~db/migrate~~ ne sont pas anodins, et qu'ils sont préfixés par le numéro de version du schéma qu'ils représentent. En fait, la tâche ~~db:migrate~~ est bien plus puissante que ce que nous avons vu. En effet, elle vous permet de migrer vers la version de votre choix de votre schéma. Pour cela, il faut cependant être précis. Jusqu'à maintenant, nous nous sommes contentés de dire ce qu'il devait se passer quand nous passions à une version supérieure du schéma. C'est la raison pour laquelle nous avons placé notre appel à ~~create_table~~ dans le corps de la méthode ~~up~~ de la classe ~~ContactInformations~~ (dans le fichier ~~002_contact_informations.rb~~ – faut suivre !). Nous avons laissé de côté la méthode ~~down~~. Encore une fois, vous l'aurez compris, sous la méthode ~~up~~, nous décrivons les modifications de schéma devant avoir lieu pour passer de la version précédente à la version courante du schéma. La méthode ~~down~~ doit décrire exactement l'inverse. Dans notre exemple, pour passer de la version 2 à la version 1, il faut supprimer les colonnes ~~nom~~, ~~prenom~~, ~~email~~ et ~~telephone~~. Pour décrire cela, nous allons utiliser la méthode ~~remove_column~~ en lui passant en argument le nom de la table que nous voulons modifier et le nom de la colonne à supprimer. Voici le nouveau code de ~~002_contact_informations.rb~~ :

```
class ContactInformations < ActiveRecord::Migration
  def self.up
    create_table :contacts, :force => true do |t|
      t.column :nom,        :string
      t.column :prenom,     :string
      t.column :email,      :string
      t.column :telephone,  :string
    end
  end

  def self.down
    remove_column :contacts, :nom
    remove_column :contacts, :prenom
    remove_column :contacts, :email
    remove_column :contacts, :telephone
  end
end
```

Si nous voulons maintenant repasser notre schéma en version 1, il suffit de faire la chose suivante :

```
$ rake db:migrate VERSION=1
(in /Users/greg/Desktop/address_book)
== ContactInformations: reverting ============================================
-- remove_column(:contacts, :nom)
   -> 0.1785s
-- remove_column(:contacts, :prenom)
   -> 0.1243s
```

```
-- remove_column(:contacts, :email)
   -> 0.2462s
-- remove_column(:contacts, :telephone)
   -> 0.1576s
== ContactInformations: reverted (0.7078s) ================================

$ sqlite3 db/address_book_development.sqlite3 <<< .dump
BEGIN TRANSACTION;
CREATE TABLE schema_info (version integer);
INSERT INTO "schema_info" VALUES(1);
CREATE TABLE contacts ("id" INTEGER PRIMARY KEY NOT NULL);
COMMIT;
```

C'est gagné ! Je vous laisse repasser seul en version 2.

4. ActiveRecord

Passons maintenant au fichier ~~app/models/contact.rb~~ généré lorsque nous avons demandé la création du modèle contact. Si vous ouvrez ce fichier avec votre éditeur préféré, vous constaterez qu'il ne comprend qu'une simple déclaration de la classe ~~Contact~~ héritant de la classe ~~ActiveRecord::Base~~. Ne vous y fiez pas. En effet, la classe ~~Contact~~ bénéficie de toute la puissance d'~~ActiveRecord~~. Et grâce à elle, vous aller pouvoir manipuler les données de la table contact très simplement. Pour illustrer cela, nous allons démarrer une console Rails :

```
$ ruby script/console
Loading development environment.
>>
```

Nous nous retrouvons sous un shell Ruby dans lequel tous les objets Rails sont accessibles, y compris ceux que nous avons créés pour notre application. Nous pouvons donc utiliser la classe ~~Contact~~. Nous allons ajouter un nouveau contact dans notre table. Il suffit simplement de créer un nouvel objet de type ~~Contact~~ en passant en paramètre au constructeur les valeurs des champs de la table, sous forme de hachage :

```
>> >> c = Contact.new(
?>     :nom => "Lejeune",
?>     :prenom => "Gregoire",
?>     :telephone => "0123456789",
?>     :email => "gregoire.lejeune@free.fr"
>>   )
=> #<Contact:0x34aee8c @new_record=true, @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@free.fr"}
>>
```

Attention
Ne vous méprenez pas, nous n'avons encore rien modifié au niveau de la base de données. Nous nous sommes contentés de créer un nouvel objet ~~Contact~~ en précisant les valeurs de ses différents attributs. Pour faire l'INSERT, au sens SQL, il faut sauver cet objet dans la base :

```
>> c.save
=> true
>>
```

Maintenant, nous avons ajouté notre enregistrement dans la table. Nous pouvons vérifier cela en demandant à retrouver tous les enregistrements de la table ~~contacts~~ :

```
>> c2 = Contact.find( :all )
=> [#<Contact:0x34a3ca8 @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@free.fr"}]
>>
```

La méthode de classe ~~find~~ renvoie un tableau d'objets ~~Contact~~. Si vous voulez récupérer le champ prénom du premier enregistrement, il suffit de faire :

```
>> puts c2[0][:prenom]
Gregoire
=> nil
```

Avec ~~find~~, nous pouvons être beaucoup plus fin que dans l'exemple précédent. Par exemple, nous pouvons remplacer ~~:all~~ par ~~:first~~, ce qui retournera non plus tous les contacts stockés dans la table, mais uniquement le premier. La notion de " premier " doit être comprise ici dans le sens où ~~find~~ travaille par défaut sur les clés primaires. En effet, vous pouvez remplacer ~~:all~~ ou ~~:first~~ par un ID. Ainsi, la commande ~~Contact.find(3)~~ renverra l'enregistrement d'ID 3.

À cela, peuvent s'ajouter des conditions à appliquer lors de la recherche. Ceci est fait en utilisant l'option ~~:conditions~~.

```
>> Contact.find( :all, :conditions => "nom like 'lej%'" )
=> [#<Contact:0x349e168 @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@free.fr"}]
>>
```

Comme vous pouvez le voir, l'écriture des conditions utilise une syntaxe SQL. Il existe de nombreuses autres options qui vous permettront d'être aussi fin que vous en avez besoin. Cependant, dans certains cas, vous pourrez avoir besoin de faire des requêtes complexes avec jointures et autres plaisirs. Bien entendu, vous pouvez vous amuser à trier, croiser… vous-même les données issues de plusieurs recherches simples. Mais, vous pouvez faire plus simple grâce à la méthode ~~find_by_sql~~. Cette méthode prend en argument une requête SQL, aussi complexe que nécessaire.

Pour la modification d'un enregistrement, nous utiliserons la méthode ~~update~~ d'~~ActiveRecord~~. Elle prend deux paramètres : le premier est l'id de l'enregistrement à modifier dans la table, le second est un hachage décrivant les modifications :

```
>> Contact.update( 1, { :telephone => "9876543210", :email => "gregoire.lejeune@gmail.com" } )
=> #<Contact:0x3487b70 @errors=#<ActiveRecord::Errors:0x34872d8 @errors={}, @base=#<Contact:0x3487b70 ...>, attributes{"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1"
>> c2 = Contact.find( :all )
=> [#<Contact:0x347b1a4 @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"9876543210", "email"=>"gregoire.lejeune@gmail.com"}]
>>
```

Dans cet exemple, nous avons considéré ne pas avoir d'objet ~~Contact~~. Ce qui sera pourtant souvent le cas. En effet, il est fréquent dans une application, de vouloir rechercher un ou plusieurs enregistrements, puis de les mettre à jour ensuite. Ceci est bien entendu possible avec ~~ActiveRecord~~ :

```
>> c = Contact.find( :first )
=> #<Contact:0x3492674 @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@free.fr"}
>> c.email = "gregoire.lejeune@gmail.com"
=> "gregoire.lejeune@gmail.com"
>> c.save
=> true
>> c = Contact.find( :first )
=> #<Contact:0x3485514 @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@gmail.com"}
>>
```

Enfin, pour supprimer un enregistrement, il faut appeler la méthode de classe ~~delete~~ en lui passant en paramètre l'id de l'enregistrement à supprimer :

```
>> Contact.delete( 1 )
=> 1
>> Contact.find( :all )
=> []
>>
```

Il est, bien entendu, possible de supprimer directement un enregistrement déjà en notre possession en lui appliquant la méthode ~~destroy~~ :

```
>> c = Contact.find( 1 )
=> #<Contact:0x347c0cc @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@gmail.com"}
>> c.destroy
=> #<Contact:0x347c0cc @attributes={"nom"=>"Lejeune", "prenom"=>"Gregoire", "id"=>"1", "telephone"=>"0123456789", "email"=>"gregoire.lejeune@gmail.com"}
>>
```

Soyez absolument certain que ce que nous venons de voir ne représente qu'une toute petite partie des possibilités offertes par ~~ActiveRecord~~. C'est cependant suffisant pour la suite de cet article.

5. Scaffolding

Revenons à notre application. Nous avons un modèle, mais toujours aucune interface humainement utilisable ! Nous avons déterminé que nous voulions pouvoir ajouter, modifier, supprimer et lister des contacts. En restant classique, et donc en oubliant le Web 2.0 pour le moment, nous pouvons donc dire qu'il nous faut autant de pages. Et encore, la suppression ne nécessite pas une interface à elle toute seule. Si nous savons que chaque page est décrite dans une vue, il faut donc créer 3 vues. Nous avons également besoin d'un contrôleur dans lequel placer la logique de notre application.

Tout ceci peut être fait en utilisant le script ~~generate~~. C'est ce que nous allons faire, mais en utilisant un superbe raccourci grâce au scaffolding. Le principe est simple, il s'agit de demander à Rails de créer pour nous tout ce que nous venons de décrire. Il suffit simplement de faire la chose suivante :

```
 ruby script/generate scaffold contact
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/contacts
      exists  app/views/layouts/
      exists  test/functional/
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
   identical    app/models/contact.rb
   identical    test/unit/contact_test.rb
   identical    test/fixtures/contacts.yml
      create  app/views/contacts/_form.rhtml
      create  app/views/contacts/list.rhtml
      create  app/views/contacts/show.rhtml
      create  app/views/contacts/new.rhtml
      create  app/views/contacts/edit.rhtml
      create  app/controllers/contacts_controller.rb
      create  test/functional/contacts_controller_test.rb
      create  app/helpers/contacts_helper.rb
      create  app/views/layouts/contacts.rhtml
      create  public/stylesheets/scaffold.css
$
```

Si vous regardez ce qui vient de se passer, vous noterez que Rails a généré un contrôleur (~~app/controllers/contacts_controller.rb~~) et quatre vues :

- ~~app/views/contacts/list.rhtml~~ présente la liste des contacts enregistrés dans notre base.
- ~~app/views/contacts/new.rhtml~~ permet de créer un nouveau contact.
- ~~app/views/contacts/edit.rhtml~~ permet d'éditer un contact existant.

- app/views/contacts/show.rhtml permet de visualiser les informations d'un contact.

Eh bien voilà, notre application est terminée ! Nous pouvons démarrer le serveur Rails et utiliser notre nouveau chef-d'œuvre :

```
$ ruby script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Starting Mongrel listening at 0.0.0.0:3000
** Starting Rails with development environment...
** Rails loaded.
** Loading any Rails specific GemPlugins
** Signals ready.  TERM => stop.  USR2 => restart.  INT => stop (no restart).
** Rails signals registered.  HUP => reload (without restart).  It might not work well.
** Mongrel available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Dans votre navigateur, connectez-vous à l'adresse http://localhost:3000/contacts et réjouissez-vous. Si vous comptez bien, vous venez de faire votre première application avec Ruby on Rails en tapant 4 commandes, en modifiant 3 fichiers et en écrivant 10 lignes de code. Les explications sur les migrations ne comptent pas ;)

Remarquez pourtant que le résultat n'est pas très beau. En effet, le scaffolding traduit en français, c'est du prototypage ! Donc, nous venons de prouver que le modèle est correct, que cela fonctionne comme on l'espérait, mais il faut maintenant rendre tout cela joli. C'est sans doute la raison pour laquelle on a inventé les web designers me direz-vous... Oui et non. En effet, dans des cas très simples, comme c'est le cas avec notre application, un peu de travail de CSS et nous pouvons avoir quelque chose de tout à fait acceptable. Certes, très proche de ce que l'on faisait il y a encore 5 ans, mais utilisable tout de même. Dans la " vraie " vie, il ne faut pas abuser du scaffolding et ne l'utiliser qu'à des fins de validation des concepts. Vous pouvez à la rigueur vous en servir comme base, mais vous constaterez souvent que votre code final sera très loin de ce qu'il était à l'origine.

Reprenons donc tout depuis le début et faisons une belle application.

6. ActionPack

Nous avons besoin de trois pages : une pour ajouter un contact, une pour modifier un contact et une affichant le contenu de notre carnet d'adresses. Pour la partie " intelligente " de l'application, nous utiliserons un contrôleur.

Pour mettre en place le squelette de ces éléments, nous allons encore une fois utiliser le script generate en lui précisant que nous voulons un contrôleur, que nous appellerons gestion_contacts, et qu'il comprendra quatre actions :

- list pour gérer la liste des contacts ;
- add pour l'ajout d'un contact ;
- modify pour la modification d'un contact ;
- delete pour la suppression d'un contact.

Nous passons donc en paramètre du script generate, le nom du contrôleur et les quatre actions :

```
      $ ruby script/generate controler gestion_contacts list add modify delete
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/gestion_contacts
    exists  test/functional/
    create  app/controllers/gestion_contacts_controller.rb
    create  test/functional/gestion_contacts_controller_test.rb
    create  app/helpers/gestion_contacts_helper.rb
    create  app/views/gestion_contacts/list.rhtml
    create  app/views/gestion_contacts/add.rhtml
    create  app/views/gestion_contacts/modify.rhtml
    create  app/views/gestion_contacts/delete.rhtml
$
```

Nous nous retrouvons avec cinq fichiers essentiels :

- app/controllers/gestion_contacts_controller.rb qui correspond au contrôleur lui-même ;
- app/views/gestion_contacts/list.rhtml est le fichier de rendu, la vue associée à l'action list du contrôleur ;
- app/views/gestion_contacts/add.rhtml est la vue associée à l'action add ;
- app/views/gestion_contacts/modify.rhtml est la vue associée à l'action modify ;
- app/views/gestion_contacts/delete.rhtml est la vue associée à l'action delete.

Je passerai sous silence test/functional/gestion_contacts_controller_test.rb, mais je pense que vous aurez compris que c'est dans ce fichier que nous pouvons écrire les tests fonctionnels de notre contrôleur. Je ne présenterai rien sur le sujet dans cet article. Il n'est cependant pas inutile de se documenter sur le sujet ;)

Si vous démarrez le serveur (ruby script/server) et que vous vous connectez à l'adresse http://localhost:3000/gestion_contacts, vous devriez avoir un message vous indiquant qu'il n'existe pas d'action index. En effet, nous n'en avons déclaré aucune. Mais avant cela, certains pourraient se demander pourquoi nous ne nous sommes pas tout simplement connectés à l'adresse http://localhost:3000. Très bonne question ! Mais, ne brûlons pas les étapes.

Dans Rails, nous accédons aux contrôleurs et aux actions associées via le format d'URL suivant :

```
http://<mon_serveur>:<mon_port>/<controller_name>/<action>
```

Dans nos exemples, le serveur étant local et le port étant celui par défaut dans Rails, nous utiliserons des adresses du type :

```
http://localhost:3000/<controller_name>/<action>
```

Comme nous l'avons remarqué, si <action> n'est pas précisé, c'est l'action index qui sera prise par défaut. À chaque action est associée une vue qui permet d'en afficher le rendu. Par défaut, la vue a le même nom que l'action. Ainsi, si nous appelons l'URL http://localhost:3000/mon_controleur/mon_action, le contrôleur utilisé sera celui codé dans le fichier app/controllers/mon_controleur.rb, dans ce contrôleur, nous demandons à appeler l'action mon_action et le rendu sera fait via la vue décrite dans le fichier app/views/mon_controleur/mon_action.rhtml.

Si nous revenons donc à notre problème d'index, vous aurez donc compris que nous avons besoin d'ajouter une action dans notre contrôleur. Éditons le fichier du contrôleur :

```
class GestionContactsController < ApplicationController
  def list
  end

  def add
  end

  def modify
  end
end
```

Vous l'avez compris, le contrôleur est une classe héritant de ApplicationController, et chaque action est définie par une méthode de cette classe. Pour ajouter l'action index, soit nous ajoutons une méthode de même nom, soit nous imposons le fait que, par défaut, nous afficherons toujours la liste des contacts, auquel cas, la méthode index peut être un alias de la méthode liste.

Si nous choisissons la dernière solution, et que nous ajoutons donc la ligne suivante après la déclaration de la méthode list :

```
alias_method :index, :list
```

Nous allons être confrontés à un nouveau problème… En effet, il nous manque la vue correspondante ; et il serait dommage de devoir dupliquer le fichier list.rhtml. Ce n'est donc pas la bonne solution. En fait, il est préférable d'ajouter l'action index, mais de préciser dans le corps de la méthode que nous voulons appeler l'action list et que nous voulons utiliser la vue list.rhtml pour le rendu. Voici le code de notre action :

```
def index
  list
  render :action => "list"
end
```

Nous venons simplement ici, d'appeler l'action list (2ème ligne), puis de préciser, via la méthode render, que nous voulons utiliser le même rendu que celui utilisé par l'action list (3ème ligne).

Il existe une autre solution, consistant simplement à faire une redirection automatique vers l'action list. Pour cela, il suffit de modifier le corps de la méthode index de la façon suivante :

```
def index
  redirect_to :action => "list"
end
```

La distinction entre ces deux solutions se comprend quand on sait que la méthode render n'est à considérer que pour un même contrôleur. En effet, il est impossible de demander à utiliser un rendu pour une action d'un autre contrôleur. Avec redirect_to, heureusement, vous pouvez préciser l'action, mais également le contrôleur :

```
redirect_to :controller => "controleur", :action => "action"
```

Vous pouvez même omettre de préciser l'action, la redirection se fera alors automatiquement vers index. De plus, vous pouvez préciser des paramètres, comme vous le feriez avec un appel d'URL classique… Je vous laisse regarder la documentation…

L'affichage de la liste des contacts étant l'action par défaut, nous allons donc commencer par coder cette action et la vue correspondante. Nous avons simplement besoin de la liste des contacts et de l'afficher. La récupération des contacts est simple, et, avec ce que nous avons vu d'ActiveRecord, vous devriez déjà être en mesure d'écrire le code de l'action :

```
class GestionContactsController < ApplicationController
  def list
    @contacts = Contact.find( :all )
  end

  # ...

end
```

Nous récupérons un tableau d'objets contacts dans la variable d'instance @contacts. C'est cette variable d'instance que nous utiliserons dans la vue pour afficher la liste. De manière générale, il faut savoir que toute variable d'instance déclarée dans une action est utilisable dans la vue servant au rendu.

Dans la vue, nous allons donc parcourir le tableau de Contact et afficher les informations contenues dans les différents attributs. Pour faire cela, il faut savoir qu'un fichier rhtml est en fait un fichier HTML embarquant du code Ruby selon la syntaxe ERB [5]. Donc, il s'agit d'HTML dans lequel le code Ruby est placé entre <% et %> et le contenu d'une expression peut être affichée si l'expression est placée entre <%= et %>. Voici donc le code de la vue list.rhtml :

```
<html>
  <head>
    <title>Liste des contacts</title>
  </head>
  <body>
    <h1>Liste des contacts</h1>

    <table>
      <tr><th>Nom</th><th>Prénom</th><th>eMail</th><th>Téléphone</th>
      <th> </th></tr>
      <% @contacts.each do |contact| %>
        <tr>
          <td><%= contact.nom %></td>
          <td><%= contact.prenom %></td>
          <td><%= contact.email %></td>
          <td><%= contact.telephone %></td>
          <td>
            <%= link_to "Modifier", :action => "modify", :id => contact %>
            <%= link_to "Supprimer", :action => "delete", :id => contact %>
          </td>
        </tr>
      <% end %>
    </table>
    <%= link_to "Ajouter un contact", :action => "add" %>
  </body>
</html>
```

La seule nouveauté ici, à moins bien entendu que vous n'ayez pas lu les premiers articles de ce magazine, est l'utilisation de ~~link_to~~. Vous l'aurez compris, nous l'utilisons pour créer un lien. L'option ~~:action~~ permet de préciser l'action vers laquelle pointe le lien. ~~:id~~ permet, quant à elle, de passer en paramètre de l'URL l'objet que nous voulons faire traiter par l'action du lien. Nous verrons un peu plus tard que ce que nous recevons est en fait l'ID (au sens de la clé primaire dans la base) de l'objet en question. Si nous avions eu besoin de passer à un nouveau contrôleur, nous aurions pu le préciser via l'option ~~:controller~~. En l'absence de cette option, Rails considère que nous continuons à travailler avec le même contrôleur. Le premier paramètre correspond, quant à lui, au titre du lien.

Si vous regardez le code HTML généré (en affichant la source de la page avec votre navigateur), vous constaterez que la ligne

```
<%= link_to "Modifier", :action => "modify", :id => contact %>
```

a été transformée en :

```
<a href="/gestion_contacts/modify/1">Modifier</a>
```

Pour l'enregistrement d'ID 1.

Ajoutez, via la console Rails, quelques contacts dans la base et regardez le résultat. Je vous propose d'ajouter une CSS pour améliorer le décor. La CSS est un fichier statique. Dans une application Rails, ce type de fichier est placé dans le répertoire ~~public~~ de l'application, en général dans le sous-répertoire ~~stylesheets~~. Créons donc un fichier ~~public/stylesheets/style.css~~. Je vous propose d'y ajouter le code suivant :

```
body {
 width: 40em;
 margin: auto;
 font-family: sans-serif;
 text-align: justify;
 font-size: 1em;
 color: #000000;
}

table {
  border: 1px solid black;
  cell-spacing: 0;
  border-spacing: 0;
  padding: 0;
}

th {
  background-color: red;
  color: white;
  font-weight: bolder;
  padding: 5px;
}

td {
  padding: 5px;
}

tr.odd {
  background-color: #eeeeee;
}

tr.even {}
```

Je n'entrerai pas dans les détails. Vous noterez simplement que nous avons déclaré deux styles pour le tag TD. En effet, il peut être assez sympathique de présenter les lignes de la table alternativement blanches (style ~~even~~) et grises (style ~~odd~~). Pour mettre en place cela dans le fichier ~~list.rhtml~~, nous allons utiliser la méthode ~~cycle~~ d'~~ActionView~~ en modifiant la ligne 13 de la façon suivante :

```
...
      <% @contacts.each do |contact| %>
```

```
        <tr class="<%= cycle('even', 'odd') %>">
          <td><%= contact.nom %></td>
...
```

Il faut aussi ajouter le chargement de la CSS. Pour cela, nous allons utiliser la méthode ~~stylesheet_link_tag~~ prenant en paramètre le nom de la CSS, sans extension :

```
...
    <title>Liste des contacts</title>
    <%= stylesheet_link_tag "style" %>
  </head>
...
```

Cette méthode, d'~~ActionView~~, génère le code HTML suivant :

```
<link href="/stylesheets/style.css?1191874047" media="screen" rel="Stylesheet" type="text/css" />
```

Maintenant que nous pouvons lister nos contacts, voyons comment en ajouter un. Commençons par la vue. Nous avons besoin d'un formulaire permettant de saisir les informations. Là encore, nous allons utiliser différentes méthodes d'~~ActionView~~. La première d'entre elles est ~~form_tag~~ qui, sur le même modèle que ~~link_to~~, permet de créer un formulaire en précisant vers quelle action et quel contrôleur doivent être renvoyées les informations saisies. De même, dans le formulaire lui-même, plutôt que d'utiliser des tags HTML, Rails nous offre différentes méthodes du genre ~~hidden_field~~, ~~password_field~~, ~~radio_button~~, ~~text_area~~, ~~text_field~~, ~~file_field~~. Mieux encore, sachez qu'il existe des méthodes pour la saisie des dates et/ou heures avec ~~date_select~~, ~~time_select~~... Voyons le code de notre vue :

```
<html>
  <head>
    <title>Ajouter un contact</title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>
    <h1>Ajouter un contact</h1>
    <% form_tag :action => 'add' do %>
      <p>Nom : <%= text_field 'contact', 'nom'  %></p>

     <p>Prénom : <%= text_field 'contact', 'prenom'  %></p>

    <p>Email : <%= text_field 'contact', 'email'  %></p>

      <p>Téléphone : <%= text_field 'contact', 'telephone'  %></p>

      <%= submit_tag "Ajouter" %>
    <% end %>
  </body>
</html>
```

Le code est clair, les données saisies dans le formulaire seront envoyées vers l'action ~~add~~ du contrôleur courant. Mais regardons attentivement le code pour la mise en place des champs texte. La méthode ~~text_field~~ prend deux paramètres obligatoires, le premier est le nom de l'objet et le second, le nom de la méthode. Pour le moment, cela n'a aucun sens. Alors, regardons ce que cela donne en HTML pour le champ de saisie du nom :

```
<input id="contact_nom" name="contact[nom]" size="30" type="text" />
```

Je sais, nous ne sommes pas plus avancé. En fait, il faut savoir que dans ~~add~~, les paramètres seront récupérables via le hachage ~~params~~. Dans le cas de notre formulaire, nous aurons, entre autres, une clé ~~contact~~, correspondant donc au nom de l'objet, dont la valeur sera elle-même un hachage, dont les clés seront les noms de méthode de chaque champ avec pour valeur, celle saisie par l'utilisateur. Si nous saisissons un utilisateur Pierre Dupond avec le mail pierre.dupond@labas.com et 0101010101 comme téléphone, ~~params~~ aura alors la " tête " suivante :

```
{"commit"=>"Ajouter",
 "contact"=>
  {"nom"=>"Dupond", "prenom"=>"Pierre", "telephone"=>"0101010101", "email"=>"pierre.dupond@labas.com"},
 "action"=>"add",
 "controller"=>"gestion_contacts"}
```

Comme vous pouvez le voir, nous avons également une clé ~~action~~ donnant l'action vers laquelle le formulaire a été envoyé, et son contrôleur (clé ~~controlleur~~), ainsi que la valeur du bouton submit (clé ~~commit~~). Ainsi, nous pourrons facilement passer les valeurs nécessaires lors de la création de l'objet ~~Contact~~. En effet, il suffira de faire un simple :

```
nouveau_contact = Contact.new( params[:contact] )
```

Mais avant cela, il faut gérer le fait que nous utilisons la même action pour appeler le formulaire et pour envoyer les données saisies. En fait, si nous appelons le formulaire sans paramètre (~~params~~ est vide), nous devons afficher le formulaire. Si ~~params~~ est rempli, nous créons un nouvel objet ~~Contact~~ et nous le sauvons dans la base avant de retourner à la liste. Nous pouvons même pousser jusqu'à faire en sorte que si l'enregistrement dans la base échoue, alors nous représentons le formulaire en gardant les données saisies par l'utilisateur.

```
Codons l'action :
          def add
    if params.has_key?( :contact )
      @contact = Contact.new(params[:contact])
      if @contact.save
        redirect_to :action => 'list'
      else
        render :action => 'add'
```

```
      end
    end
  end
```

Et la reprise des données par le formulaire me direz-vous… Eh bien, ne modifiez rien… Vous allez voir, c'est de la magie ! Et pour vous en rendre compte, passons à la modification.

Pour la vue, nous allons (pour le moment) nous contenter de reprendre le code de la vue ~~add.rhtml~~, en remplaçant les différents libellés " Ajouter " par " Modifier " :

```
<html>
  <head>
    <title>Modifier un contact</title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>
    <h1>Modifier un contact</h1>
    <% form_tag :action => 'add', :id => @contact do %>
      <p>Nom : <%= text_field 'contact', 'nom'  %></p>

      <p>Prénom : <%= text_field 'contact', 'prenom'  %></p>

      <p>Email : <%= text_field 'contact', 'email'  %></p>

      <p>Téléphone : <%= text_field 'contact', 'telephone'  %></p>

      <%= submit_tag "Modifier" %>
    <% end %>
  </body>
</html>
```

Pour l'action, nous avons besoin de vérifier si ~~params~~ a une clé ~~:contact~~ et, si tel est le cas, nous mettons à jour les données dans la table en utilisant la méthode ~~update_attributes d'ActiveRecord~~. Si ~~params~~ n'a pas de clé ~~:contact~~ ou si la mise à jour des données échoue, nous affichons la vue ~~modify.rhtml~~, sinon nous retournons à la liste :

```
  def modify
    @contact = Contact.find(params[:id])
    if params.has_key?( :contact )
       and @contact.update_attributes(params[:contact])
      redirect_to :action => 'list'
    end
  end
```

Vous remarquerez que nous n'avons rien ajouté au niveau des ~~text_field~~ du formulaire. Essayez pour voir… C'est pas magique ça ? Eh oui, en précisant un objet et une méthode dans ~~text_field~~, Rails affecte automatiquement à chaque champ la valeur de l'attribut correspondant à sa méthode, prise dans l'objet. Donc pour

```
text_field 'contact', 'prenom'
```

Rails a ajouté la valeur de l'attribut ~~prenom~~ de l'objet contenu dans ~~@contact~~… Et attendez, on peut faire encore mieux. Si vous regardez les vues ~~add.rhtml~~ et ~~modify.rhtml~~, vous vous rendez compte immédiatement qu'il y a du code commun. En effet, toute la partie contenue entre les balises de création de formulaire, et à l'exception du bouton submit, est la même dans les deux. Il serait dommage, si un jour nous modifions le schéma de base en ajoutant un champ par exemple, de devoir faire deux modifications. Eh bien, soyez heureux, nous pouvons mutualiser cette partie grâce à un partial !

Un partial est une partie de vue que nous pouvons inclure dans une vue. Ils sont placés dans des fichiers dont le nom commence par un underscore (" _ "). Nous pouvons, pour notre application, créer un partial ~~_form.rhtml~~ contenant le bout de formulaire commun :

```
<p>Nom : <%= text_field 'contact', 'nom'  %></p>
<p>Prénom : <%= text_field 'contact', 'prenom'  %></p>
<p>Email : <%= text_field 'contact', 'email'  %></p>
<p>Téléphone : <%= text_field 'contact', 'telephone'  %></p>
```

Nous pouvons maintenant remplacer le code équivalent, dans ~~add.rhtml~~ et ~~modify.rhtml~~ par l'appel suivant :

```
        <%= render :partial => 'form' %>
```

À partir de maintenant, toute modification de ~~_form.rhtml~~ sera visible dans ~~add.rhtml~~ et ~~modify.rhtml~~.
Mieux encore… Vous avez certainement remarqué que nous avons du code commun entre les trois vues… Eh oui, ceci :

```
<html>
  <head>
    <title> ... TITRE ...</title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>

    # Code spécifique

  </body>
</html>
```

Le seul élément " gênant " est le titre. Nous lui jetterons un sort plus tard. Sachez que là encore, nous pouvons mutualiser. Il suffit de créer un layout pour le contrôleur. Un layout est un fichier rhtml ayant le même nom qu'un contrôleur et que l'on place dans le répertoire ~~app/views/layouts~~ de notre application. Ce fichier est utilisé pour le rendu de chaque vue du contrôleur, sauf demande express, en indiquant à quel endroit doit être ajouté le code de la vue. Nous pouvons donc créer un fichier ~~app/views/layouts~~

/gestion_contacts.rhtml avec le code suivant :

```
<html>
  <head>
    <title><%= @page_title %></title>
    <%= stylesheet_link_tag "style" %>
  </head>
  <body>

    <%= yield  %>

  </body>
</html>
```

L'instruction <%= yield %> indique l'appel au fichier de rendu de la vue attachée à l'action. Pour le titre, nous avons résolu le problème en utilisant une variable de classe @page_title qu'il suffit donc d'ajouter dans le code des actions au niveau du contrôleur. Simple, mais efficace. Vous pouvez maintenant modifier les différents fichiers de vues pour ne garder que ce qui se trouve entre les balises <body> et </body>.

Il nous reste une dernière action à traiter : la suppression d'un contact. Dans ce cas, nous n'avons pas besoin de vue. En effet, l'utilisateur clique sur le lien " Supprimer " en face d'un contact, nous appelons l'action delete et nous réaffichons la liste :

```
  def delete
    Contact.find(params[:id]).destroy
    redirect_to :action => 'list'
  end
```

Simple, efficace, mais dangereux. En effet, si l'utilisateur clique par erreur, il n'y a aucun garde-fou ! Il serait en effet sympathique de demander à l'utilisateur de confirmer sa demande. Inutile de créer une vue pour cela. Nous allons simplement modifier l'affichage du lien " Supprimer " de la liste en demandant à Rails d'ajouter une boite de confirmation. Voici le bout de code modifié dans list.rhtml :

```
...
            <%= link_to "Modifier", :action => "modify", :id => contact %>
            <%= link_to "Supprimer",
              { :action => 'delete', :id => contact },
              :confirm => 'Etes vous sure de vous ?',
              :method => :post %>
          </td>
...
```

Et voici le code HTML généré correspondant :

```
        <a href="/gestion_contacts/delete/1" onclick="if (confirm('Etes vous sure de vouloir effacer ce contact ?')) { var f = document.createElement('form'); f.style.d
```

Je ne vous avais pas menti, Rails vous simplifie la vie !

7. Bonne Route !

Tout à l'heure, certains se sont demandés pourquoi, et surtout comment, nous pouvons utiliser une URL simple pour accéder à notre application. En effet, il serait beaucoup plus agréable de se connecter via l'URL http://localhost:3000 plutôt que via http://localhost:3000/<contrôleur>/<action>. Pire, si vous essayez de vous connecter à votre application sans préciser (au minimum) le contrôleur, vous arrivez sur une page vous signalant que vous êtes sur un site Rails, sans plus. Si vous fouillez les sources de notre application, vous retrouverez ce fichier sous le nom index.html dans le répertoire public.

La première chose à faire est simplement de supprimer ce fichier. Ensuite, il faut modifier les " routes " de notre application. Les routes sont gérées dans le fichier config/routes.rb du répertoire de notre application.

Ce fichier permet de préciser le comportement du serveur face à une URL donnée. Dans notre cas, nous avons besoin de préciser que s'il n'y a aucun chemin précisé dans l'URL, nous voulons rediriger l'utilisateur vers le contrôleur gestion_contacts. Nous ajoutons donc la ligne suivante dans le fichier des routes :

```
        map.connect '', :controller => "gestion_contacts"
```

Vous l'aurez compris, la définition d'une route est similaire à celle d'une redirection (avec redirect_to)
Vous pouvez vous amuser à en ajouter d'autres. Mais restez logique ;)

8. Pagination

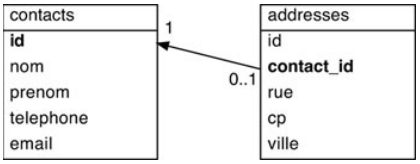
Si vous revenez sur le code généré avec scaffold, vous remarquerez vite qu'il est plus riche que ce que nous venons de faire. En effet, il présente les solutions existantes pour faire de la pagination par exemple. C'est un des éléments pour lesquels je vous conseille de faire attention si vous commencez à travailler avec Rails 2.0. En effet, dans la future version du framework, ces méthodes ont été écartées de Rails Core au profit de plugins. Je vous conseille de regarder will_paginate[6] pour vous laisser séduire.

9. Retour sur ActiveRecord

L'application que nous venons de mettre en place est très simple. Trop simple pour nous permettre d'aborder toutes les possibilités que nous offre Rails. Il y a cependant un domaine que je ne peux pas passer sous silence et qui concerne ActiveRecord.

Dans notre exemple, nous n'avons travaillé qu'avec une seule table. C'est une exception. En effet, il est très rare d'avoir des applications aussi simples. Sans entrer dans le développement, nous allons donc voir les solutions qui existent quand nous avons besoin de gérer des relations 1:1, 1:N et N:N dans une application Rails.

La relation 1:1 est extrêmement simple. Elle est représentée dans la base de données par la présence d'une clé étrangère dans une table, pointant vers une et une seule entrée d'une autre table. Imaginons pour cela que nous ajoutions dans notre application une table d'adresses. Nous voulons pouvoir, pour chaque contact, lui ajouter une adresse. J'ai pleinement conscience qu'une personne peut avoir plusieurs adresses et que, inversement, une adresse peut être partagée par plusieurs personnes. Mais, oublions cela et imposons que dans notre application une adresse n'est liée qu'à une seule personne, et inversement. La question que nous devons nous poser maintenant consiste à savoir qui est rattaché à qui ? Donc, si la clé étrangère doit se trouver dans la table adresse, cequi implique qu'une adresse pointe vers un contact ou si c'est l'inverse. Une chose est certaine, il ne peut pas y avoir d'adresse sans contact. Cela n'aurait aucun sens. Par contre, l'inverse est possible. La solution la plus logique pour construire notre nouveau schéma est donc la suivante :



Pour ajouter la table adresses, nous allons commencer par créer un nouveau modèle :

```
$ ruby script/generate model address
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/address.rb
      create  test/unit/address_test.rb
      create  test/fixtures/addresses.yml
      exists  db/migrate
      create  db/migrate/003_create_addresses.rb
$
```

Il nous suffit ensuite de modifier le fichier de migration 003_create_addresses.rb :

```
class CreateAddresses < ActiveRecord::Migration
  def self.up
    create_table :addresses, :force => true do |t|
      t.column :contact_id, :integer
      t.column :rue,        :string
      t.column :cp,         :string
      t.column :ville,      :string
    end
  end

  def self.down
    drop_table :addresses
  end
end
```

N'oubliez pas de faire la migration avec un rake db:migrate !

Nous avons donc deux modèles dans notre application : Contact et Address. Pour décrire la relation existant entre ces modèles, nous les modifierons de la façon suivante :

```
contact.rb :

class Contact < ActiveRecord::Base
  has_one :address
end

address.rb :

class Address < ActiveRecord::Base
  belongs_to :contact
end
```

Nous venons simplement de préciser qu'un contact a une adresse (~~has_one~~) et qu'une adresse se réfère à un contact (~~belongs_to~~). Ces modifications vont nous ouvrir la voie à tout un monde de simplicité. En effet, nous pouvons maintenant très facilement ajouter une adresse à un contact :

```
na = Address.new( ... )

c = Contact.new( ... )
c.address = na
```

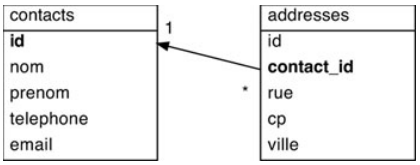
Si, par la suite, nous récupérons un contact, nous pouvons récupérer son adresse, sous forme d'objet ~~Address~~, tout aussi simplement :

```
c = Contact.find( :first )
addresse_du_contact = c.address
```

L'inverse est également possible. Si vous recherchez une adresse, vous pouvez retrouver le contact qui y habite de la façon suivante :

```
a = Address.find( :first )
contact_a_cette_adresse = a.contact
```

C'en est presque trop simple !
Comme nous l'avons dit, il arrive qu'une personne ait plusieurs adresses. Nous allons donc transformer notre relation 1:1 en relation 1:N dans ce sens. Dans ce cas, la présence de la clé étrangère dans la table addresses reste parfaitement logique. En effet, une adresse est bien liée à une et une seule personne, mais une personne peut avoir plusieurs adresses. Voici le schéma correspondant :



Cette modification peut paraître mineure, elle le sera, tout au moins dans notre code... En effet, la seule chose que nous devons préciser, c'est qu'un contact peut avoir plusieurs adresses. Il suffit pour cela de remplacer dans ~~contact.rb~~ le ~~has_one~~ par un ~~has_many~~ :

```
class Contact < ActiveRecord::Base
  has_many :addresses
end
```

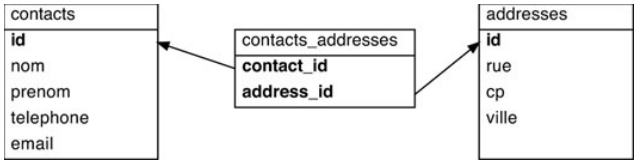
Notez que nous avons pluralisé ~~address~~. En haut, le ~~has_one :address~~ est devenu ~~has_many :addresses~~. C'est logique, puisqu'il y en a plusieurs !
Si maintenant vous recherchez les adresses d'un contact, alors qu'avec la relation 1:1 vous n'aviez qu'un objet de type ~~Address~~ pour un contact, vous obtiendrez maintenant un tableau d'objets ~~Address~~ :

```
c = Contast.find( :first )
c.addresses
  # => [#<Address:0xNNNNNNN @attributes={ ... }, ...]
```

Pour ajouter une adresse à un contact, il suffit d'ajouter une nouvelle entrée dans sa table d'adresses :

```
c = Contact.find( :first )
nouvelle_adresse = Address.new( ... )
c.addresses << nouvelle_address
c.save
```

Encore une fois, c'est enfantin !
Pour être, enfin, tout à fait exhaustif sur les cas possibles, nous devons considérer que non seulement une personne peut avoir plusieurs adresses, mais qu'il peut y avoir plusieurs personnes à une même adresse. Nous devons donc mettre en place une relation de type N:N entre nos deux tables. Ce type de relation ne peut pas se faire sans une table de jointure pour aboutir au schéma suivant :



Nous devons donc ajouter une table ~~contacts_addresses~~. Pour faire cela, nous allons encore faire une migration :

```
$ ruby script/generate migration lien_nn_contacts_addresses
      exists  db/migrate
      create  db/migrate/004_lien_nn_contacts_addresses.rb
$
```

Dans le fichier ~~db/migrate/004_lien_nn_contacts_addresses.rb~~, nous allons demander à créer la table ~~contacts_addresses~~ et nous allons également supprimer la clé étrangère de la table ~~addresses~~. Voici le résultat :

```
class LienNnContactsAddresses < ActiveRecord::Migration
  def self.up
    create_table :contacts_addresses, :id => false, :force => true do |t|
      t.column :contact_id, :integer
      t.column :address_id, :integer
    end

    remove_column :addresses, :contact_id
  end

  def self.down
    drop_table :contacts_addresses

    add_column :addresses, :contact_id
  end
end
```

La seule chose importante à noter ici est la présence de l'option ~~:id => false~~ lors de la demande de création de la table de jointure ~~contacts_addresses~~. Cette option permet de préciser que la table ne doit pas avoir de clé primaire. Vous pouvez lancer la migration.

Il faut maintenant modifier les classes ~~Contact~~ et ~~Address~~ pour préciser que non seulement un contact peut être lié à plusieurs

adresses, mais qu'une adresse peut, elle aussi, être liée à plusieurs contacts. Ceci est fait en remplaçant ~~has_many~~ et ~~belongs_to~~, respectivement dans ~~Contact~~ et ~~Address~~, par ~~has_and_belongs_to_many~~ dans les deux classes :

```
class Contact < ActiveRecord::Base
  has_and_belongs_to_many :addresses
end

class Address < ActiveRecord::Base
  has_and_belongs_to_many :contact
end
```

Nous pouvons maintenant ajouter plusieurs adresses à un contact et inversement, l'attribut ~~adresses~~ de la classe ~~Contact~~ étant un tableau d'objets ~~Address~~ et l'attribut ~~contacts~~ de la classe ~~Address~~ étant un tableau d'objets ~~Contact~~.

Conclusion

Vous avez certainement remarqué que je n'ai absolument pas parlé de MVC. J'ai certes cité les termes "Modèle", "Vue" et "Contrôleur", mais sans approfondir les concepts. Eh bien, je n'en dirai pas plus. En fait, il n'est absolument pas nécessaire d'être un expert MVC pour pouvoir développer avec Rails. Il suffit simplement de savoir que les modèles sont définis dans ~~app/modeles~~, que les vues vont dans ~~app/views~~ et que les contrôleurs vont dans ~~app/controllers~~. Pour le reste, il ne s'agit que de règles de développement. Si vous souhaitez en savoir plus, je vous conseille de lire l'article sur Camping...

Références:

- [1] http://www.basecamphq.com
- [2] http://rubyonrails.org
- [3] http://www.yaml.org
- [4] http://api.rubyonrails.org
- [5] http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/
- [6] http://railscasts.com/episodes/51

~~Retrouvez cet article dans : Linux Magazine Hors série 33~~

Posté par (La rédaction) | Signature : Grégoire Lejeune | Article paru dans 

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

« Précédent Aller au contenu »
Identifiez-vous
Inscription
S'abonner à UNIX Garden

Catégories

- Administration réseau
- Administration système
- Agenda-Interview
- Audio-vidéo
- Bureautique
- Comprendre
- Distribution
- Embarqué
- Environnement de bureau
- Graphisme
- Jeux
- Matériel
- News
- Programmation
- Réfléchir
- Sécurité
- Utilitaires
- Web

Il y a actuellement

994 articles/billets en ligne.

GO

Archives

Articles secondaires

15/3/2009
Smart Middle Click 0.5.1 : ouvrez les liens JavaScript dans des onglets

Tout d'abord, un petit raccourci utile : quand vous voulez ouvrir un lien dans un onglet, plutôt que d'utiliser le menu contextuel, cliquez simplement dessus avec le bouton du milieu. Hop, c'est ouvert ! C'est simple et diablement efficace, parfois un peu trop……

Voir l'article...
30/10/2008
Google Gears : les services de Google offline

Lancé à l'occasion du Google Developer Day 2007 (le 31 mai dernier), Google Gears est une extension open source pour Firefox et Internet Explorer permettant de continuer à accéder à des services et applications Google, même si l'on est déconnecté.…

Voir l'article...
7/8/2008
Trois questions à...

Alexis Nikichine, développeur chez IDM, la société qui a conçu l'interface et le moteur de recherche de l'EHM.…

Voir l'article...
11/7/2008
Protéger une page avec un mot de passe

En général, le problème n'est pas de protéger une page, mais de protéger le répertoire qui la contient. Avec Apache, vous pouvez mettre un fichier ~~.htaccess~~ dans le répertoire à protéger.…

Voir l'article...
6/7/2008
hypermail : Conversion mbox vers HTML

Comment conserver tous vos échanges de mails, ou du moins, tous vos mails reçus depuis des années ? mbox, maildir, texte… les formats ne manquent pas. …

Voir l'article...
6/7/2008
iozone3 : Benchmark de disque

En fonction de l'utilisation de votre système, et dans bien des cas, les performances des disques et des systèmes de fichiers sont très importantes.…

Voir l'article...

Articles de 1ère page

- Journée Méditerranéenne des Logiciels Libres 2008

- Les dessous d'Android
- GNU/Linux Magazine HS N°44 - OCTOBRE/NOVEMBRE 2009 - Chez votre marchand de journaux
- MISC N°45 - SEPTEMBRE/OCTOBRE 2009 - Chez votre marchand de journaux
- Encore plus loin avec OpenLDAP
- Interview de Luke Kanies
- GNU/Linux Magazine N°119 - SEPTEMBRE 2009 - Chez votre marchand de journaux
- Linux Pratique N°55 - Septembre/Octobre 2009 - Chez votre marchand de journaux
- Les sysadmins jouent à la poupée
- Instrumentation scientifique reconfigurable