

Tutoriel / Mars 2006

Créez votre blog avec Rails !

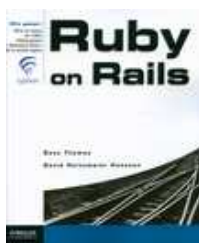


Richard Piacentini

Valdateur technique pour l'adaptation française du livre *Agile Web Development With Rails: A Pragmatic Guide*, à savoir [Ruby on Rails](#) dans sa version française, et créateur du portail [Railsfrance.org](#) (<http://www.railsfrance.org>), [Richard Piacentini](#) vous propose de réaliser une petite application de [création de blog](#) avec Ruby on Rails.

D'autres parties seront ajoutées à ce tutoriel pour compléter cette application et la rendre totalement opérationnelle. Ainsi, n'hésitez pas à revenir régulièrement sur ce tutoriel. Mais avant de vous plonger dans la création de votre blog, voici un peu de théorie !

Qu'est ce que Ruby on Rails ?



[Ruby on Rails](#)

Ruby on Rails est un framework open-source pour le développement d'applications web, écrit en langage Ruby, et entièrement basé sur le **modèle MVC** (Modèle-Vue-Contrôleur). Au premier abord rien de bien extraordinaire me direz-vous, car il existe un grand nombre de frameworks, écrits dans différents langages et basés eux aussi sur ce fameux modèle MVC.

Rails présente cependant l'avantage de fournir toutes les couches de l'architecture et de les faire travailler ensemble de manière totalement intégrée. Qui plus est, Rails a été conçu afin de respecter au mieux les deux concepts suivants : "Ne vous répétez pas" (DRY) et "Convention plutôt que configuration".

Le résultat final est un framework intégral permettant de développer des applications web robustes et évolutives en moins de lignes de code qu'il n'en faut pour le dire et de maximiser la productivité et le plaisir de coder de l'heureux développeur qui l'utilise... D'ailleurs certains aficionados affirment qu'on reconnaît un développeur Rails à l'indéfectible sourire affiché en permanence sur son visage radieux.

Mais comme vous n'êtes pas non plus obligés de me croire sur parole, je vous propose de réaliser une petite [application de blog](#) afin de faire connaissance avec celui qui deviendra bientôt votre meilleur outil pour le développement d'applications web : Ruby on Rails.

Installer Ruby on Rails sous Windows et OS X

Pour les utilisateurs de Windows le plus simple est d'utiliser le Ruby One-Click Installer [1], la version utilisée pour ce tutoriel est la 1.8.4-16 preview3. Une fois l'installation de Ruby terminée, ouvrez un shell DOS et lancez la commande `gem install rails --include-dependencies -r`.

Pour les utilisateurs de Mac OS X 10.4 Ruby est déjà installé mais hélas la version 1.8.2 livrée par Apple pose des problèmes de fonctionnement avec Rails. Vous pouvez soit corriger le problème comme expliqué ici [2], soit installer Locomotive [3] ou encore télécharger ce script [4] qui installera tout le nécessaire.

Que vous soyez sous Windows ou Mac OS X installez aussi la librairie RedCloth car nous utiliserons Textile pour le formatage du corps de nos articles et commentaires. Il est fortement recommandé d'utiliser la version 3.0.3 avec Rails 1.0, ouvrez donc un terminal et tapez cette commande : `gem install redcloth --version "3.0.3"`

- [1] <http://rubyforge.org/projects/rubyinstaller/>
- [2] <http://tech.rufy.com/articles/2005/05/01/complete-fix-for-ruby-on-mac-os-x-10-4-tiger>
- [3] <http://locomotive.raaum.org/home/>
- [4] <http://nubyonrails.topfunky.com/articles/2005/12/29/an-even-better-way-to-build-ruby-rails-lighttpd-and-mysc>

Création de notre future application de blog

Pour cela il suffit de lancer la commande `rails nom_de_l_application`, et notre framework préféré se chargera de générer le squelette de l'application en plaçant judicieusement les fichiers qui la composent dans différents répertoires que nous explorerons au fur et à mesure de notre progression dans ce tutoriel.

Ouvrez une fenêtre DOS ou lancez votre shell favori, placez-vous à l'endroit où vous souhaitez créer la structure de l'application et lancez cette commande :

```
rails demoblog
```

Si tout se passe bien vous devriez voir quelque chose dans ce genre :

- create
- create app/controllers
- create app/helpers
- create app/models
- create app/views/layouts
- create config/environmentss
- create components
- create db
- create doc
- create lib
- create lib/tasks
- create log
- create public/images
- create public/javascripts
- create public/stylesheets
- create script/performance
- create script/process
- create test/fixtures
- create test/functional
- create test/mocks/development
- create test/mocks/test
- create test/unit
- create vendor
- create vendor/plugins
- create Rakefile
- create README
-

- create log/server.log
- create log/production.log
- create log/development.log
- create log/test.log

Placez-vous maintenant dans le nouveau répertoire **demoblog**, si vous jetez un coup d'oeil sous le répertoire **app/** vous verrez que Rails a déjà créé les répertoires **model**, **views** et **controllers** où seront stockés les différents fichiers correspondants à nos modèles, vues et, rien de surprenant, contrôleurs. Un autre répertoire nommé **helpers** se trouve au même niveau mais nous y reviendrons en temps utile.

Explorons maintenant le répertoire **script/**, dans lequel vous trouverez différents utilitaires. Celui qui nous intéresse pour le moment est le fichier nommé **server**. Ce dernier permet de lancer un serveur web autonome (WEBrick ou lighttpd si il a été installé au préalable) et de profiter de notre application nouvellement créée. Essayez donc en tapant la commande suivante :

```
ruby script/server
```

Ensuite lancez votre navigateur web et rendez-vous à l'adresse suivante :

```
http://127.0.0.1:3000/
```

Voici ce qui devrait s'afficher :

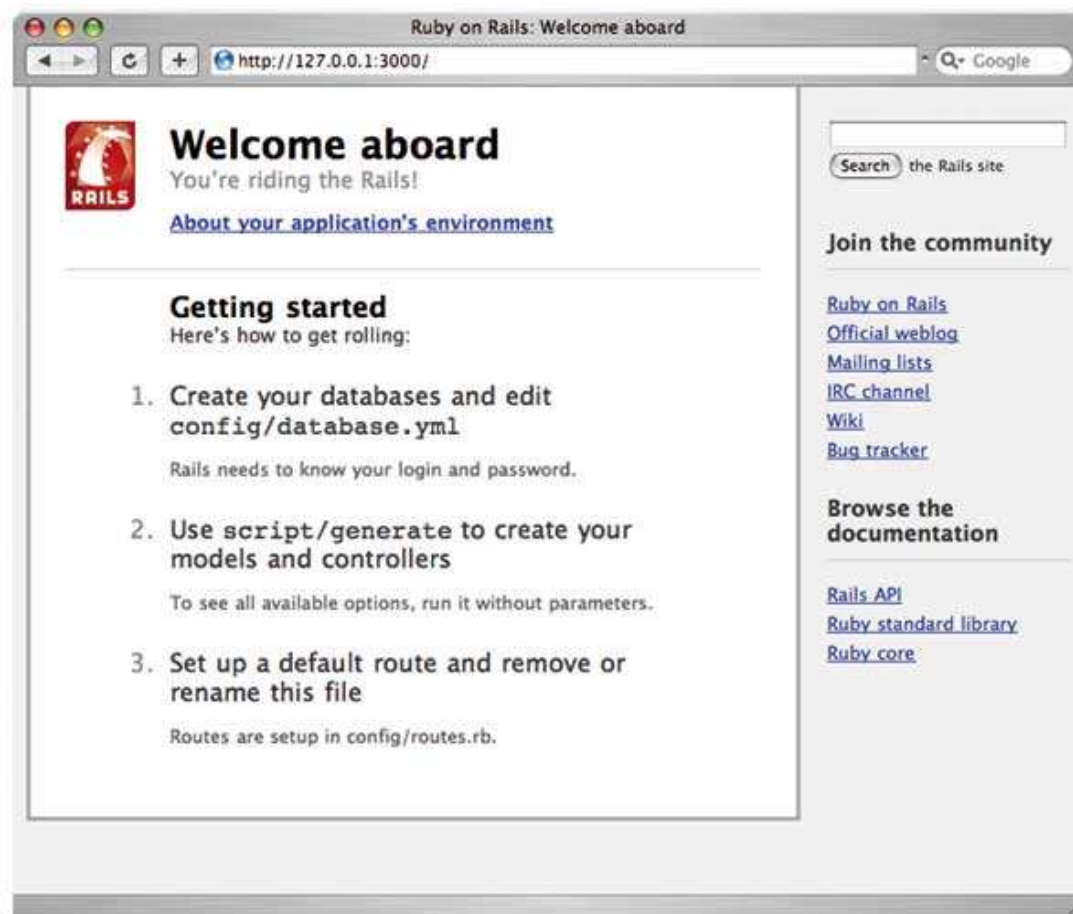


Figure 1 : Bienvenue à bord

Comme vous pouvez le constater Rails fait le maximum afin que nous puissions entrer au plus vite dans le vif du sujet. Le simple fait de taper deux commandes (rails demoblog et ruby script/server) a non seulement généré la structure de notre application mais a aussi mis en place un environnement nous permettant d'y accéder immédiatement via notre navigateur web !

J'entends d'ici les plus véhéments d'entre vous me rétorquer que tout ceci est bien beau mais que pour le moment nous sommes encore bien loin de pouvoir devenir les nouvelles étoiles de la planète blog, et qu'il serait peut être

opportun de penser à utiliser une base de données pour y stocker les passionnants articles que nous ne manquerons pas d'écrire pour la postérité... C'est justement ce que nous allons faire dès la section suivante.

Configuration de la base de données à utiliser

Munissez-vous de votre éditeur de texte favori et éditez le fichier **config/database.yml**. Vous remarquerez que Rails a déjà mis en pratique sa fameuse philosophie "Convention plutôt que configuration". En effet, le nom de la base de données est déjà pré-renseigné à partir du nom de notre application.

Les plus attentifs d'entre vous auront néanmoins remarqué qu'il n'y a pas une mais trois entrées, **development:**, **production:** et **test:**, correspondant à trois bases de données *demoblog_development*, *demoblog_test* et *demoblog_production*. Pendant la phase de développement de notre application, c'est la base *demoblog_development* qui sera utilisée.

La base *demoblog_test* quant à elle est considérée comme temporaire et pourra être réinitialisée à chaque lancement de la batterie de tests que nous ne manquerons pas d'écrire, en bons développeurs consciencieux que nous sommes. Nous verrons d'ailleurs plus loin que, fidèle à son habitude, Rails devance une fois de plus nos attentes et nous a déjà préparé toute une infrastructure de test.

Pour finir la base de données *demoblog_production* sera utilisée lorsque notre application sera véritablement mise en ligne.

Si vous utilisez **MySQL** la seule chose qu'il vous reste à faire à ce niveau, c'est de renseigner les champs *username* et *password* de la section **development:**. Nous nous occuperons plus tard de la section **test:**. Maintenant, créez la base de données *demoblog_development* avec la commande en ligne `mysql`, `phpMyAdmin`, `CocoaMySQL` ou tout autre outil que vous utilisez habituellement.

Si vous utilisez **SQLite**, mettez en commentaire ou supprimez la section par défaut concernant MySQL, et renommez la section correspondant à votre version de SQLite en **development:** (et si ce n'est pas déjà fait installez le driver Ruby comme expliqué dans les commentaires du fichier *config/database.yml*). Il est inutile de créer la base de données SQLite à l'avance, Rails le fera automatiquement lorsque nous générerons le schéma de la base de données.

Pour les autres bases de données le mode opératoire est le même. Commentez ou supprimez la section par défaut concernant MySQL, renommez la section correspondant à votre version en **development:** et configurez la si besoin.

Une fois la configuration de la base de données terminée il est impératif de **redémarrer le serveur web** pour que les modifications soient prises en compte.

Création du schéma de la base de données

Tout d'abord nous allons générer un fichier schéma vide pour notre application, tapez donc la commande suivante :

```
rake db_schema_dump
```

Si vous voyez apparaître ce message d'erreur :

- rake aborted!
- Unknown database 'demoblog_development'

C'est probablement parce-que vous n'utilisez pas SQLite et que vous n'avez pas encore créé la base de données... Relancez la commande une fois que ce sera fait.

Un nouveau fichier nommé *schema.rb* se trouve maintenant dans le répertoire **db/**, voici ce qu'il contient :

schema.rb(vide)

- # This file is autogenerated. Instead of editing this file, please use the
- # migrations feature of ActiveRecord to incrementally modify your
- # database, and

- # then regenerate this schema definition.
- ActiveRecord::Schema.define() do
- end

Nous allons le renseigner et définir notre schéma en utilisant le langage Ruby. Le code SQL compatible avec la base de données que nous avons configuré dans le fichier *db/database.yml*, sera automatiquement généré par Rails au moment de l'importation.

Evidemment rien ne vous empêche de créer le schéma avec vos outils habituels (phpMyAdmin, CocoaMySQL, etc.) mais ce serait beaucoup moins portable car lié au type de base de données que vous utilisez au moment de la création.

Ne vous inquiétez pas, même si il s'agit de votre première rencontre avec Ruby, vous allez vous rendre compte que c'est extrêmement lisible. Vous n'avez qu'à recopier le fichier ci-dessous.

schema.rb(final)

- # This file is autogenerated. Instead of editing this file, please use the
- # migrations feature of ActiveRecord to incrementally modify your
- # database, and
- # then regenerate this schema definition.
- ActiveRecord::Schema.define() do
- create_table "articles" do |t|
- t.column "titre", :string, :limit => 100
- t.column "texte", :text
- t.column "created_at", :datetime
- t.column "updated_at", :datetime
- t.column "commentaires_count", :integer, :default => 0
- end
- create_table "commentaires" do |t|
- t.column "texte", :text
- t.column "created_at", :datetime
- t.column "updated_at", :datetime
- t.column "article_id", :integer
- end
- end

Notre application de blog utilise deux tables de base de données, une pour stocker nos passionnants articles et l'autre pour les commentaires qui ne manqueront pas d'affluer de toutes parts. Ces tables sont nommées *articles* et *commentaires*, **au pluriel** et en minuscule, respectez bien cette convention car elle a son importance comme nous le verrons plus loin.

La table *articles* est composée d'un champ nommé *titre* de type chaîne (:string) limité à 100 caractères, et d'un autre champ nommé *texte*. Tout cela est très classique jusqu'ici.

Les trois autres champs néanmoins sont plus intéressants car directement liés à notre fameux adage "Convention plutôt que configuration". En effet les deux champs de type datetime nommés *created_at* et *updated_at* seront automatiquement renseignés par Rails avec la date et l'heure de création ou de modification de l'enregistrement. Le champ *commentaires_count* quant à lui va permettre à Rails de stocker le nombre de commentaires rattachés à un article sans aucun effort de notre part, juste parce qu'il porte le nom de la table *commentaires* suivi du suffixe *_count* (notez qu'il est important que ce champ soit de type entier (integer) avec une valeur par défaut de zéro).

Dans la table *commentaires* le champ *article_id* est la clef externe qui va permettre de rattacher les commentaires à l'article auquel ils appartiennent. Là aussi, le nommage est important puisque ce champ porte le nom, **au singulier**, de la table à laquelle il fait référence suivi du suffixe *_id*.

La clef primaire de chacune des tables est générée par Rails et se nomme *id* par défaut. Il s'agit là d'une convention que vous pouvez outrepasser si besoin est.

Pour importer le schéma dans la base de données il nous suffit maintenant de taper la commande suivante :
rake db_schema_import.

Nous pouvons maintenant passer à la section suivante.

Génération des modèles, vues et du contrôleur

Dans une application développée avec Rails le modèle représente la plupart du temps une table de la base de données.

ActiveRecord est la couche de mise en correspondance objet-relationnel (ORM) de Rails, son mode de fonctionnement est très proche du modèle ORM standard : les tables de la base de données correspondent aux classes du modèle, les lignes des tables correspondent aux objets et les colonnes aux attributs de ces objets. ActiveRecord se différencie néanmoins des autres bibliothèques ORM par l'utilisation de conventions de nommage (encore elles...) qui permettent généralement de limiter la configuration.

Dans le cadre de cet article nous ne détaillerons pas le mode de fonctionnement d'ActiveRecord, mais si vous désirez aller plus loin sachez que, pour commencer, il est indispensable de comprendre les conventions utilisées pour la correspondance entre les noms des classes du modèle et les tables de la base de données. Vous trouverez une documentation abondante à ce sujet sur Internet, et pour les possesseurs du livre [Ruby on Rails](#), vous trouverez des informations dans le chapitre 14 intitulé *Les bases de ActiveRecord*.

Par défaut ActiveRecord suppose que le nom de la table correspond à la forme plurielle du nom de la classe, et si la classe comporte des mots commençants par une lettre majuscule, ils seront transformés en minuscule et séparés par un tiret bas. Ainsi, la classe nommée *Article* est mise en correspondance avec la table nommée *articles*, la classe nommée *ArticleAuteur* avec la table *article_auteurs* et la classe *RoleUtilisateur* avec la table *role_utilisateurs*.

Sachez tout de même que l'algorithme utilisé pour déterminer la forme plurielle des noms des tables est simpliste et ne fonctionnera pas pour la majorité des formes irrégulières de la langue française. Néanmoins cela reste de la convention, et dans Rails toute convention peut être remplacée par de la configuration, en cas de besoin, vous pouvez donc désactiver ce mode de fonctionnement et fournir explicitement le nom de la table à associer à un modèle.

Refermons donc cette parenthèse conceptuelle et revenons-en à notre application, il est temps désormais de créer les modèles correspondants à nos tables *articles* et *commentaires*, ils seront donc conventionnellement nommés *Article* et *Commentaire*.

Utilisons pour cela le script `generate` qui est un outil très pratique que vous utiliserez souvent lors du développement de vos applications Rails.

ruby script/generate model Article

- exists app/models/
- exists test/unit/
- exists test/fixtures/
- create app/models/article.rb
- create test/unit/article_test.rb
- create test/fixtures/articles.yml

Vous pouvez remarquer que Rails encourage vraiment les bonnes pratiques car en plus du fichier modèle (*app/models/article.rb*), il met en place l'environnement qui nous permettra de tester efficacement les composants de notre future application. C'est dans le fichier *test/unit/article_test.rb* que nous pourrons écrire les **tests unitaires** portants sur notre modèle et le fichier de garniture *test/fixtures/articles.yml* permettra de spécifier le contenu initial qui sera chargé automatiquement dans la base de données de test.

De la même manière générons le modèle *Commentaire* :

ruby script/generate model Commentaire

- exists app/models/
- exists test/unit/
- exists test/fixtures/
- create app/models/commentaire.rb
- create test/unit/commentaire_test.rb
- create test/fixtures/commentaires.yml

Maintenant préparez-vous à atteindre des sommets de productivité, car en une seule ligne de commande nous allons générer un contrôleur pour notre modèle *Article*, les méthodes permettant de créer, récupérer, mettre à jour et supprimer des articles (le fameux CRUD en anglais : Create, Retrieve, Update, Delete) ainsi que les vues associées à ces actions.

Comble d'originalité, nous allons baptiser notre contrôleur *Blog* et la commande ci-dessous va générer tout l'échafaudage ("scaffold" en anglais) de notre application. D'ailleurs nous aurions très bien pu nous passer de générer préalablement le modèle *Article* et laisser le "scaffolding" s'en charger.

```
ruby script/generate scaffold Article Blog
```

- exists app/controllers/
- exists app/helpers/
- create app/views/blog
- exists test/functional/
- dependency model
- exists app/models/
- exists test/unit/
- identical app/models/article.rb
- identical test/unit/article_test.rb
- identical test/fixtures/articles.yml
- create app/views/blog/_form.rhtml
- create app/views/blog/list.rhtml
- create app/views/blog/show.rhtml
- create app/views/blog/new.rhtml
- create app/views/blog/edit.rhtml
- create app/controllers/blog_controller.rb
- create test/functional/blog_controller_test.rb
- create app/helpers/blog_helper.rb
- create app/views/layouts/blog.rhtml
- create public/stylesheets/scaffold.css

Tapez l'url suivante dans votre navigateur web `http://127.0.0.1:3000/blog` et voici ce qui devrait s'afficher sous vos yeux ébahis :



Figure 2 : Liste articles

Et si vous cliquez sur le lien *New article* vous accéderez immédiatement au formulaire vous permettant de créer des articles :

The screenshot shows a web browser window titled "Blog: new" with the address bar containing "http://127.0.0.1:3000/blog/new". The main content area displays a form titled "New article". The form includes a "Titre" (Title) input field, a "Texte" (Text) text area, and two date pickers labeled "Created at" and "Updated at", both showing "2006", "March", "1", "02", and "22". Below the date pickers are "Create" and "Back" buttons.

Figure 3 : Formulaire de création

Impressionnant non ? Vous disposez en un temps record d'une interface web immédiatement fonctionnelle. Tous les champs définis dans notre table *articles* sont présents sur le formulaire, et vous pouvez déjà créer, éditer ou supprimer des articles ! Essayez par vous-même et, vous allez vous rendre compte que notre application de Blog fonctionne réellement.

Si vous ouvrez maintenant le fichier *app/controllers/blog_controller.rb*, vous constaterez que Rails a généré tout le code nécessaire au bon fonctionnement de notre interface, libre à nous ensuite de le modifier et de le faire évoluer en fonction de nos besoins. Rails nous fournit tous les éléments pour démarrer et être opérationnel le plus vite possible mais ce n'est pas une fin en soi, bien au contraire.

Création des relations entre les modèles et validation des données

Pour commencer nous allons définir explicitement la relation existant entre le modèle *Commentaire* et le modèle *Article*, à savoir qu'un commentaire "appartient" à un article.

Pour cela, éditez le fichier *app/models/commentaire.rb* et recopiez ceci :

-
- `class Commentaire < ActiveRecord::Base`
 - `belongs_to :article, :counter_cache => true`
 - `validates_presence_of :texte`
 - `end`

La déclaration *belongs_to* signifie que la table qui contient la clef étrangère (*commentaires*) "appartient" à la table qu'elle référence (*articles*). En pratique cela signifie pour Rails que la table *commentaires* contient une colonne nommée *article_id* qui référence la colonne *id* dans la table *articles*. Le nom de la clef étrangère (*article_id*) est fixé par convention et peut, comme toujours, être modifié en cas de besoin.

Le fait de rajouter `:counter_cache => true` indique à ActiveRecord de maintenir un compteur pour le nombre de commentaires associés à chaque article, c'est le champ *commentaires_count* de la table *articles* qui sera utilisé à cet effet. Si vous ajoutez un commentaire le compteur sera incrémenté, si vous supprimez un commentaire le compteur sera alors décrémenté, et le tout de manière totalement automatique.

Enfin la ligne `validates_presence_of :texte` demande à Rails de vérifier que le champ nommé "texte" est bien renseigné avant de sauvegarder le commentaire dans la base de données. Dans le cas contraire le formulaire de saisie sera réaffiché avec un message d'erreur.

En pratique un article peut avoir plusieurs commentaires qui lui sont associés, modifions donc le fichier *app/models/article.rb* en conséquence :

- `class Article < ActiveRecord::Base`
- `has_many :commentaires, :dependent => true`
- `validates_presence_of :titre, :texte`
- `end`

Nous avons donc rajouté la déclaration `has_many :commentaires` dont l'option `:dependent => true` signifie que les commentaires ne peuvent exister indépendamment de l'article auquel ils font référence. En l'occurrence la destruction d'un article entraînera automatiquement la destruction des commentaires associés.

De la même manière que pour les commentaires, le fait d'ajouter `validates_presence_of :titre, :texte` nous assure que Rails vérifiera que ces champs sont renseignés avant qu'un article puisse être sauvegardé. Voici ce qui apparaîtra si vous essayez de sauver un article sans avoir correctement renseigné les champs obligatoires :

Figure 4 : Erreur de saisie

Et voilà, nous pouvons maintenant passer à l'étape suivante.

Modification de la mise en page de notre application

Il faut reconnaître que l'habillage de notre application est pour le moment plutôt austère et nous allons y remédier immédiatement.

Dans l'archive contenant le code source de l'application finale vous trouverez dans *public/images* toutes les **images** utilisées (copiez les dans votre répertoire *demoblog/public/images*) ainsi que la **feuille de style** *public/stylesheets/blog.css* à copier dans votre répertoire *demoblog/public/stylesheets/* en remplacement de *scaffold.css* que nous n'utiliserons pas.

Pour commencer nous allons revoir la mise en page générale de notre application. Il existe plusieurs moyens de définir des mises en page dans Rails mais sachez que si vous placez dans le répertoire *app/views/layouts/* un fichier portant le même nom que le contrôleur alors toutes les vues générées par ce contrôleur utiliseront par défaut cette mise en page.

Un fichier de mise en page *app/views/layouts/blog.rhtml* a déjà été créé lors de la génération de l'échafaudage (scaffolding) de notre application. Nous allons maintenant le remplacer par ce qui suit :

- `<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"`
- `"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">`
- `<html xmlns="http://www.w3.org/1999/xhtml">`
- `<head>`
- `<meta http-equiv=Content-Type content="text/html; charset=utf-8" />`

- <title>Rails Blog</title>
- <%= stylesheet_link_tag "blog" %>
- <%= javascript_include_tag "prototype", "effects" %>
- </head>
- <body>
- <div id="page">
- <div id="top">
- <div id="titre_blog">
- A la découverte de Ruby on Rails
- </div>
- </div><!-- fin div top -->
- <div id="main">
- <div id="content">
- <p style="color: green"><%= flash[:notice] %>
- <%= @content_for_layout %>
- </div><!-- fin div content -->
- <div id="footer">propulsé par ReFRESH</div>
- </div><!-- fin div page -->
- </body>
- </html>

Une fois cette opération effectuée voici ce que vous devriez voir apparaître dans votre navigateur :

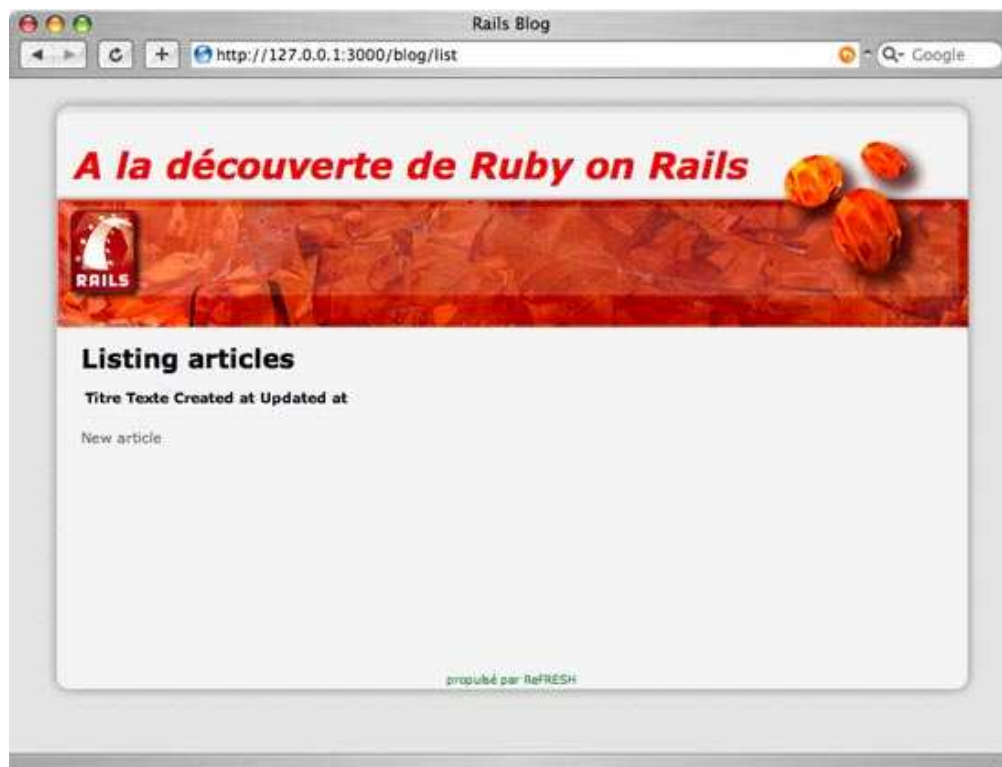




Figure 5a & 5b : Notre nouvelle mise en page

Habillage des pages d'ajout et d'édition d'articles

Modifions maintenant les vues `app/views/blog/new.rhtml` et `app/views/blog/edit.rhtml`, remplacez leur contenu par celui-ci :

new.rhtml

- `<h2 class="new">Ecrire un nouvel article</h2>`
- `<div id="form-content">`
- `<%= start_form_tag :action => 'create' %>`
- `<%= render :partial => 'form' %>`
- `<%= submit_tag "Envoyer" %>`
- `<%= end_form_tag %>`
- `</div>`
- `<%= link_to image_tag('retour.gif'), :action => 'list' %>`

edit.rhtml

- `<h2 class="edit">Editer l'article</h2>`
- `<div id="form-content">`
- `<%= start_form_tag :action => 'update', :id => @article %>`
- `<%= render :partial => 'form' %>`
- `<%= submit_tag 'Editer' %>`
- `<%= end_form_tag %>`
- `</div>`

- `<%= link_to image_tag('retour.gif'), :action => 'list' %>`

Notez qu'afin d'éviter la duplication de code nos deux vues *new.rhtml* et *edit.rhtml* utilisent un même fichier (*_form.rhtml*) pour afficher le formulaire de saisie. Il s'agit d'un fichier partiel ("partial" en anglais) dont le contenu est inséré à l'endroit où il est appelé. Il est impératif que le nom du fichier contenant le partiel commence par un tiret bas (`_`), même si ce tiret ne doit pas apparaître lors de l'appel du partiel avec la méthode **render** (`render :partial => 'form'`).

Nous allons modifier le fichier partiel *app/views/blog/_form.rhtml* afin de supprimer l'affichage des champs *created_at* et *updated_at* qui seront mis à jour automatiquement par Rails lors de la création et de la modification d'enregistrements.

_form.rhtml

- `<%= error_messages_for 'article' %>`
- `<!--[form:article]-->`
- `<p><label for="article_titre">Titre</label>
`
- `<%= text_field 'article', 'titre' %></p>`
- `<p><label for="article_texte">Texte</label>
`
- `<%= text_area 'article', 'texte' %></p>`
- `<!--[eoform:article]-->`

Affichage de la liste des articles

Passons maintenant à l'affichage de la liste des articles. Voici le contenu à copier dans le fichier *app/views/blog/list.rhtml* :

- `<p id="nouvel-article"><%= link_to image_tag('article.gif'), :action => 'new' %></p>`
- `<%= render :partial => 'article', :collection => @articles %>`
- `<div id="page_infos">`
- `<% if @article_pages.length > 1 %>`
- `<div id="page_num">`
- `<page <%= @article_pages.current.to_i %> / <%= @article_pages.length %>`
- `</div>`
- `<% end %>`
- `<div id="liens-pagination">`
- `<%= link_to(image_tag('prev.gif'), :alt => 'page précédente'), { :page => @article_pages.current.previous`
- `},`
- `{ "id" => "link-precident" }) if @article_pages.current.previous %>`
- ` `
- `<%= link_to(image_tag('next.gif'), :alt => 'page suivante'),`
- `{ :page => @article_pages.current.next },`
- `{ "id" => "link-suivant" }) if @article_pages.current.next %>`
- `</div>`
- `</div>`

Nous allons créer un nouveau fichier partiel *app/views/blog/_article.rhtml* que nous utiliserons à la fois pour l'affichage d'une liste d'articles et d'un article en particulier. Voici le contenu à copier dans ce fichier :

- `<div class="article">`
- `<div class="article_top"></div>`
- `<div class="article-message">`
- `<h2 class="article-titre">`
- `<%= link_to h(article.titre), :action => 'show', :id => article %>`
- `</h2>`
- `<div class="article-etat">`
- `<div class="article-info">`
- `<posté le <%= article.created_at.strftime("%d/%m/%Y à %H:%M:%S") %>`
- `</div>`
- `<div class="article-edit">`

- `<%= link_to image_tag('editer.gif', :alt => 'editer cet article'),`
- `{:action => 'edit' , :id => article} %>`
- `<%= link_to image_tag('supprimer.gif', :alt => 'supprimer cet article'),`
- `{:action => 'destroy' , :id => article},`
- `:confirm => "supprimer cet article ?"%>`
- `</div>`
- `</div>`
- `<div class="article-texte"><%= textilize article.texte %></div>`
- `<div class="article-commentaire">`
- `<%= link_to 'commentaires',`
- `:action => 'show',`
- `:id => article %> (<%= article.commentaires_count.to_s %>)`
- `<%= link_to image_tag('comment.jpg', :alt => 'afficher les commentaires'),`
- `:action => 'show',`
- `:id => article %>`
- `</div>`
- `</div>`
- `<div class="article_bas"></div>`
- `</div>`

Notez que cette simple ligne `<%= textilize article.texte %>` indique à Rails de transformer les marqueurs au format Textile en code HTML, ce qui est très pratique pour formater le corps de nos articles (pour que cela fonctionne il faut néanmoins que RedCloth soit installé).

Affichage d'un article et ajout de commentaires

Modifions le fichier `app/views/blog/show.rhtml` afin d'afficher un article et ses commentaires :

show.rhtml

- `<%= link_to image_tag('retour.gif', :alt => "retour à la liste"),`
- `:action => 'list' %>`
- `

`
- `<%= render :partial => 'article' %>`
- `
`
- `<div id="commentaires-list">`
- `<%= render :partial => "commentaire",`
- `:collection => @article.commentaires %>`
- `</div>`
- `<div id="commentaire_infos">`
- `<div id="commentaire_spinner">`
- `<%= image_tag('spinner.gif',`
- `:style => 'display:none',`
- `:id => 'spinner' %>`
- `</div>`
- `<div id="commentaires-erreur"></div>`
- `<div id="commentaire_action">`
- `<%= image_tag 'commenter.gif', :alt => 'ajouter un commentaire' %>`
- `</div>`
- `</div>`
- `

`
- `<%= render :partial => 'form_commentaire' %>`
- `<div id="bottom"></div>`

Si vous regardez attentivement dans le fichier de mise en page de notre application (`app/views/layouts/blog.rhtml`) vous remarquerez la présence de cette ligne qui va charger les bibliothèques JavaScript *prototype* et *effects* : `<%= javascript_include_tag "prototype", "effects" %>`

Nous aurons ensuite accès sans effort aux méthodes AJAX et à une ribambelle d'effets visuels, qui nous permettront par exemple de faire apparaître en douceur le formulaire de saisie juste en dessous du dernier commentaire affiché, le tout dans le plus pur style "web 2.0"...

Il est temps maintenant de créer deux nouveaux fichiers partiels, l'un *app/views/blog/_commentaire.rhtml* est utilisé pour l'affichage de la liste des commentaires et l'autre *app/views/blog/_form_commentaire.rhtml* pour l'affichage du formulaire d'ajout de commentaire. Voici le contenu de ces deux fichiers :

_commentaire.rhtml:

- <div class="commentaire_glob">
- <div class="commentaire_top"></div>
- <div class="commentaire-message">
- <div class="commentaire-info">
- <posté le <%= commentaire.created_at.strftime("%d/%m/%Y à %H:%M:%S") %>
- </div>
- <div class="commentaire-texte">
- <%= textilize commentaire.texte %>
- </div>
- </div>
- <div class="commentaire_bas"></div>
- </div>

_form_commentaire.rhtml

- <div id="form-content" style="display: none;">
- <h3>Ajoutez vos commentaires !</h3>
- <%= form_remote_tag(:update => {:success => 'commentaires-list',
- <:failure => 'commentaires-erreur'},
- <:url => {:action => 'ajout_commentaire',
- <:id => @article },
- <:loading => "Effect.Appear('spinner')",
- <:after => "Effect.DropOut('form-content')",
- <:complete => "\$('spinner').style.display = 'none';\$('commentaire_texte').value="; new
- <Effect.ScrollTo('bottom');return false;"
- <:position => 'bottom') %>
- <%= text_area('commentaire', 'texte', 'cols' => 40, 'rows' => 10) %>
-

- <%= submit_tag 'Envoyer' %>
- <%= end_form_tag %>
- </div>

Comme vous pouvez le constater nous utilisons AJAX pour ajouter des commentaires, ainsi que des effets visuels pour faire disparaître le formulaire de saisie et faire défiler la page... et tout ceci fait partie intégrante de Rails. Il ne nous reste plus qu'à définir la méthode *ajout_commentaire* dans notre contrôleur. Voici le code que vous pouvez copier dans *app/controllers/blog_controller.rb*, par exemple juste après la méthode *destroy* :

- def ajout_commentaire
- # le commentaire ne sera pas créé si il est vide
- unless (params[:commentaire][:texte]).empty?
- c = Article.find(params[:id]).commentaires.create(params[:commentaire])
- render :partial => "commentaire", :locals => { :commentaire => c }
- else
- render :nothing => true
- end
- end

Réglons quelques petits détails

Vous avez sans doute remarqué que dans la liste les articles sont affichés chronologiquement, les plus anciens en

premier alors que le contraire serait tout de même plus pratique. Qu'à cela ne tienne, éditez le fichier `app/controllers/blog_controller.rb` et modifiez la méthode `list` comme ci-dessous :

- `def list`
- `@article_pages, @articles = paginate :articles,`
- `:per_page => 10,`
- `:order => 'id DESC'`
- `end`

Profitons-en aussi pour modifier le routage afin d'arriver par défaut sur la page de liste des articles, pour cela supprimez le fichier `public/index.html`, puis éditez le fichier `config/routes.rb`, décommentez la ligne `# map.connect " :controller => "welcome"` et remplacez la par `map.connect " :controller => "blog"`.

Et voilà, c'est terminé !

C'est la fin de la première partie de ce tutoriel, certes notre application n'est pas encore totalement opérationnelle, en grande partie parce-que l'ajout et la suppression d'articles ne sont pas protégés mais nous verrons dans le prochain tutoriel comment mettre rapidement en place cette fonctionnalité, et bien d'autres encore.

En attendant, j'espère que cela vous aura donné envie d'aller plus loin avec Ruby on Rails. Mais avant de vous quitter je ne peux résister à l'envie de vous montrer une copie d'écran de la commande `rake stats` qui nous donne toutes sortes d'informations statistiques sur le code que nous venons de produire.

La ligne qui nous intéresse ici est *Code LOC: 62 (LOC signifie Lines Of Code)*, il s'agit du nombre de lignes de code réel dans l'application (sans tenir compte des commentaires ou du code de test) et sur ces 62 lignes nous n'avons pas du en écrire plus d'une dizaine nous mêmes... C'est cette incroyable productivité, alliée à l'expressivité du langage Ruby qui ont fait le succès de Ruby on Rails.

En attendant la suite de ce tutoriel n'hésitez pas à vous rendre sur le portail **RailsFrance** et sur le site **RubyFR** où vous trouverez tout plein d'informations concernant Rails et Ruby.

```

Terminal — bash — 80x24
ricp@enkidu:~/projects/demos/eyrolles/demoblog-partie_1$ rake stats
(in /Users/ricp/projects/demos/eyrolles/demoblog-partie_1)
+-----+
| Name           | Lines | LOC | Classes | Methods | M/C | LOC/M |
+-----+
| Helpers        | 6     | 4   | 0       | 0       | 0   | 0     |
| Controllers    | 63    | 50  | 2       | 9       | 4   | 3     |
| Components     | 0     | 0   | 0       | 0       | 0   | 0     |
|  Functional tests | 88    | 63  | 2       | 10      | 5   | 4     |
| Models         | 8     | 8   | 2       | 0       | 0   | 0     |
|  Unit tests    | 67    | 48  | 2       | 8       | 4   | 4     |
| Libraries      | 0     | 0   | 0       | 0       | 0   | 0     |
+-----+
| Total          | 232   | 173 | 8       | 27      | 3   | 4     |
+-----+
Code LOC: 62   Test LOC: 111   Code to Test Ratio: 1:1.8

ricp@enkidu:~/projects/demos/eyrolles/demoblog-partie_1$

```

Captures d'écran du blog version finale :



Propos recueillis par [Laetitia Maraninchi](#).

A découvrir

Le livre :

- [Ruby on Rails](#)

L'interview de Laurent Julliard sur :

- [le langage Ruby](#)

Les ressources en ligne :

- <http://www.rubyonrails.org/> - le site officiel (en anglais).
- <http://www.railsfrance.org/> - le portail de la communauté francophone des utilisateurs de Ruby on Rails.
- <http://rubyfr.org> – le site francophone dédié au développement avec le langage Ruby.
- <http://questionnaire.journaldunet.com/fiche/1492/3/index.html> - un sondage sur RoR

A venir dans ce tutoriel

- Protéger l'ajout d'article et les fonctions d'édition et suppression
- Modifier les messages par défaut affichés en cas d'erreur de saisie
- Rajouter des catégories et utiliser les migrations
- Rajouter un moteur de recherche live avec AJAX
- Tester les différents composants de notre application

Déjà en ligne

- [Création d'un blog I : modèle MVC, configuration de la base de données...](#)
- [Création d'un blog II : ajout et édition d'articles, ajout de commentaires...](#)

Téléchargements

- [Codes source](#)
 - [Tutoriel en version PDF](#)
-