Ruby on Rails

Ruby on Rails est un framework web MVC complet, basé sur Rack, fournissant une large gamme d'outils et une configuration par défaut, permettant un développement rapide.

Ce framework dispose d'un guide complet dont ces transparents ne sont qu'un extrait sur les points les plus importants.

installation et création d'une application

```
<pre 'code'>
gem install rails
gem install therubyracer # pour exécuter du code javascript
rails new path/to/directory
```

Description de l’arborescence

```
.
├── app                                  # tous le code de l'application
│   ├── assets                           # images, javascript et CSS
│   │   ├── images
│   │   ├── javascripts
│   │   └── stylesheets
│   ├── controllers                      # mvC
│   ├── models                           # Mvc
│   └── views                            # mVc
├── config
│   ├── database.yml                     # configuration base
│   ├── environment.rb
│   ├── environments
│   │   ├── development.rb
│   │   ├── production.rb
│   │   └── test.rb
│   └── routes.rb                        # définition du routage
├── config.ru # config Rack
├── db                                   # schémas et migration de la BD
├── doc
├── Gemfile                              # configuration des librairies
├── lib                                  # librairies hors Rails
├── log
├── public                               # répertoire public vu par le client HTTP
│   ├── 404.html
```

```
│   ├── 422.html
│   ├── 500.html
│   ├── favicon.ico
│   ├── index.html
│   └── robots.txt
├── Rakefile                                    # tâches à appeler
├── README
├── script
├── test
├── tmp
└── vendor
```

Constituants MVC de Ruby on Rails

Modèle et persistence :

Rails fournit des générateurs liés à la gestion de modèles persistent par ActiveRecord :

- `rails g[enerate] migration NAME [field[:type][:index] field[:type][:index]]`
  génère une migration dans `db/migrate`

- `rails g model NAME [field[:type][:index] field[:type][:index]]` : génère un modèle héritant de `ActiveRecord::Base` dans `app/models/name.rb`. Par défaut, appelle le générateur de migration.

Exemple :

```
rails g model Person name:string age:integer
  invoke  active_record
  create    db/migrate/20120305100527_create_people.rb
  create    app/models/person.rb
  invoke    test_unit
```

```
  create      test/unit/person_test.rb
  create      test/fixtures/people.yml
```

Modèle != Persistence != ORM BD

La logique d’une application ne nécessite pas forcément de la persistence. Une application peut disposer d’un modèle dont la gestion n’est pas liée à celle d’une base de données ou d’un autre moyen de persistence.

Routage

Le routage dans Rails est spécifié dans le fichier `config/routes.rb`

Une route permet de trouver la classe Contrôleur et la méthode de cette classe devant être exécutées pour gérer le traitement de la requête.

Pour lister les routes générées par les directives de ce fichier : `rake routes`

Règle de routage générale

```
match route
```

route doit être composée d'éléments séparés par des `/` :

- éléments statiques : une chaîne de caractères
- un paramètre : `:`chaîne de caractères.

Les paramètres `:controller` et `:action` permettent de repérer la classe contenant le contrôleur et l'action (la méthode dans la classe) à exécuter. Tous les autres paramètres sont disponibles dans la méthode à partir du

tableau associatif `params`.

```
match ':controller/:action/:id'
GET /users/show/42 : exécute users#show avec params[:id] == 42
```

On peut également spécifier le contrôleur et l’action :

```
match 'photos/:id' => 'photos#show'
GET /photos/42 : exécute photos#show avec params[:id] == 42
```

Routage pour une ressource REST

```
<pre 'code'>
#config/routes
resources :posts
```

```
<pre 'code'>
rake routes
    posts GET    /posts(.:format)          {:action=>"index", :controller=>"posts"}
          POST   /posts(.:format)          {:action=>"create", :controller=>"posts"}
 new_post GET    /posts/new(.:format)      {:action=>"new", :controller=>"posts"}
edit_post GET    /posts/:id/edit(.:format) {:action=>"edit", :controller=>"posts"}
     post GET    /posts/:id(.:format)      {:action=>"show", :controller=>"posts"}
          PUT    /posts/:id(.:format)      {:action=>"update", :controller=>"posts"}
          DELETE /posts/:id(.:format)      {:action=>"destroy", :controller=>"posts"}
```

Nommage des routes

Rails permet d'utiliser des alias à la place des URL dans le code des contrôleurs et des vues

Par exemple, avec les routes définies ci-dessus pour la ressource posts :

```
posts_path   # retourne "/posts"
posts_url    # retourne "http://server_name:port/posts"
```

On peut spécifier soi-même un alias avec l’option `:as => :nom` :

```
match 'disconnect' => 'sessions#destroy', :as => :logout
```

génère :

```
logout_path # retourne '/disconnect'
logout_url  # idem avec le nom du serveur
```

Contrôleur

Les contrôleurs contiennent le code assurant le traitement des requêtes. Chaque route configurée dans la couche de routage est associée à une méthode d'un contrôleur.

Un contrôleur dans Rails est une classe héritant de `ApplicationController`.

```
<pre 'code'>
class MyController < ApplicationController
  def action
  end
end
```

Contrôleur : Gestion des paramètres

Les paramètres provenant du routage, de la ligne de requête `?param=value` ou d'un formulaire HTML sont accessibles par le tableau associatif `params`.

Rails génère automatiquement un tableau si un paramètre multivalué a un nom finissant par []

```
GET /clients?ids[]=1&ids[]=2&ids[]=3 => params[:id] == ["1", "2", "3"]
```

Rails génère automatiquenent un tableau associatif si on rajoute en plus un identifants entre [et]

```
<pre 'code'>
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" />
  <input type="text" name="client[phone]" />
  <input type="text" name="client[address][postcode]" />
  <input type="text" name="client[address][city]" />
</form>
```

Si le formulaire est validé avec : `Acme, 06543210, 31000, Toulouse`

```
<pre 'code'>
params[:client] == {:name => "Acme", :phone => "06543210",
                    :address => {:postcode => "31000", :city => "Toulouse" }
                   }
```

Contrôleur : divers conteneurs gérés

Session

La session est automatiquement gérée est disponible par le tableau associatif `session`.

Cookies

Les cookies sont manipulables par le tableau associatif `cookies`.

Flash

Le tableau associatif `flash` permet de stocker dans la session un message qui sera effacé automatiquement après la prochaine requête.

Utilisation : stocker un message avant de faire un redirect. Le message ne sera disponible que pour la réponse au redirect.

Exemple :

1. Client demande de création de ressource : `POST /resource`
2. Serveur : `flash["message"] = "ressource 42 créée !"`, redirect

vers `/resource/42`,
3. Client : `GET /resource/42`
4. Serveur : génère une page HTML affichant le contenu de `flash["message"]`, détruit `flash["message"]`
5. Plus tard, Client : `GET /resource/42` : le HTML retourné n’affiche plus le message.

Contrôleur : filtre

```
<pre 'code'>
class MyController < ApplicationController
  before_filter :require_login, :only => [:protected_action]
  def protected_action
     # do stuff …
  end

  def require_login
    unless session[:current_user]
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  end
end
```

Contrôleur : schéma du codage d'une action

```
<pre 'code'>
class MyController < ApplicationController
  def action
    # gestion des conteneurs : session, params, flash, cookies
    # on utilise le modèle
    @model = Model.find(params[:id])  # Create, Read, Update, Delete
    # réponse complète : on génère un résultat
    render …
    # OU redirection : réponse 30
    head …

    # si aucun des 3 précisé, action par défaut : render sans argument,
    # équivalent à render "action"
    # template dans app/views/my_controller/action.html.erb
  end
end
```

render

- une vue : `render 'my_view'` : template dans `app/views/my_controller/my_view.html.erb`
- le template d'une autre action : `render 'other_action'`
- le template d'une action d'un autre contrôleur: `render 'products/show'`
- un template dans un fichier quelconque : `render '/path/to/file'`
- template inline : `render :inline => "<%= model.name %>"`
- texte : `render :text => "Toto"`
- json/xml : `render :json => @model # appelle to_json sur @model`
- javascript : `render :js => "alert('Bonjour le monde');"`

Redirection

```
redirect_to url/path
```

À la place de fournir l'URL ou le chemin, on utilise généralement les raccourcis fournis par le routage (1ère colonne de l'affichage de `rake routes`)

Vues

Les vues sont situées dans `app/views`, et regroupées par contrôleur. Les fichiers contenant les templates sont nommés `.format_resultat.moteur_template`

Par exemple : `.html.erb` indique que le moteur de template `erb` va être appelé et que le résultat sera du HTML.

Layout

Un layout est une factorisation de certains éléments du rendu. À mettre dans `app/views/layouts`.

Règle de recherche du layout pour le contrôleur `FooController` :

1. `app/views/layouts/foo.html.erb`
2. `app/views/layouts/application.html.erb` layout par défaut

On peut également spécifier le layout Ou spécifier dans l'appel de `render` :

dans le contrôleur :

<pre 'code'>
```
class FooController < ApplicationController
  layout "specific"
  #...
end
```

<pre 'code'>
```
render :layout => 'specific_layout'
render :layout => false # pas de layout
```

Écriture d’un layout

Utilisez `yield` pour faire référence à la vue. Le rendu global correspond au rendu du layout dans lequel on remplace le bloc `yield` par le rendu de la vue.

```
<pre 'code'>

<!DOCTYPE html>
<html>
<head>
  <title>Mon application</title>
</head>
<body>

<%= yield %> <!-- sera remplacé par le template de la vue -->

</body>
</html>
```

Partial

Une vue peut faire appel à une autre vue, appelée vue partielle (partial). Le nom d'un partial commentce par _. Il suffit d'appeler dans la vue principale :

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

<pre 'code'>

```
# /app/views/posts/edit.html.erb
<h1>Editing post</h1>

<%= render 'form' %>

<%= link_to 'Show', @post %> |
<%= link_to 'Back', posts_path %>
```

<pre 'code'>

```
# /app/views/posts/_form.html.erb
<%= form_for(@post) do |f| %>
  <div class="field">
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </div>
  <div class="field">
    <%= f.label :content %><br />
    <%= f.text_area :content %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

TP

Nous allons utiliser un générateur de rails pour étudier le principe de fonctionnement des différents constituants :

Scaffolding et étude du résultat

Scaffolding : génération d'un modèle, d'un contrôleur et des vues associés permettant la gestion du modèle par les actions CRUD avec des routes REST :

```
<pre 'code'>
rails new testapp # création d'application
cd testapp
# edit Gemfile, rajouter "gem 'therubyracer'"
bundle install # bundle update pour mettre à jour
rails g scaffold post title:string body:text # génère une application complète
rake db:migrate
rails server # http://localhost:3000/posts
```

Répondez aux questions suivantes :

1. Étudiez les routes générées, d'abord dans le fichier `config/routes.rb`, puis par la sortie de `rake routes`.
2. Créez un post dans l'interface. Comment est positionné le message "Post was successfully created." en haut de la page ? Rechargez la page, le message disparaît, expliquez le mécanisme.
3. Faites apparaître sur toutes les pages le nom du contrôleur et de l'action ayant généré le résultat. Pour cela, modifiez le layout.
4. Modifiez le code pour que la génération d'un post redirige vers la liste de tous les posts.
5. Étudiez la génération du formulaire de modification : listez les fichiers mis en jeu.