Ruby on Rails

2e édition

Dave Thomas

David Heinemeier Hansson

Avec la collaboration de Leon Breedt, Mike Clark,
James Duncan Davidson, Justin Gehtland et Andreas Schwarz

ISBN : 978-2-212-12079-0

EYROLLES

4

Plaisir immédiat

Écrivons une application Web triviale pour vérifier que Rails est correctement installé sur notre machine. Au passage, nous jetterons un coup d'œil à la façon dont les applications Rails fonctionnent.

Créer une nouvelle application

Quand vous installez le framework Rails, vous disposez d'une nouvelle commande, `rails`, capable de mettre en place le squelette d'une nouvelle application Rails.

Pourquoi avons-nous besoin de cet outil spécial et pourquoi ne pas simplement s'armer de notre éditeur favori pour écrire le code de notre application Rails ? En fait nous pourrions parfaitement le faire. Après tout une application Rails n'est élaborée qu'à partir de code source Ruby. Mais Rails s'occupe aussi d'un tas de choses en arrière-plan qui permettent à notre application de fonctionner avec un minimum de configuration explicite. Pour cela, Rails doit pouvoir trouver tous les composants de notre application. Comme nous le verrons plus tard (dans le chapitre 14, section *Structure des répertoires*), ceci nous amènera à créer une structure de répertoires spécifique, chaque portion de code écrite devant être rangée au bon endroit. La commande `rails` se charge simplement de mettre en place cette structure de répertoires pour nous et de l'enrichir avec du code Rails standard.

Pour créer votre première application Rails, ouvrez une fenêtre shell et placez-vous à l'endroit de votre système de fichiers où vous souhaitez créer la structure arborescente de votre application. Dans notre exemple, nous allons créer nos projets dans un répertoire appelé *`work`*. Dans ce répertoire, utilisons la commande `rails` pour créer une application que nous

appellerons `demo`. À ce stade faites attention : s'il existe déjà un répertoire *demo*, Rails vous demandera confirmation avant de détruire le répertoire existant.

```
dave> cd work
work> rails demo
create
create  app/apis
create  app/controllers
create  app/helpers
   :      :      :
create  log/development.log
create  log/test.log
work>
```

La commande a créé un répertoire *demo*. Placez-vous dans ce répertoire et listez son contenu en utilisant `ls` sous Unix/Linux ou `dir` sous Windows. Vous y trouverez une série de fichiers et de dossiers.

```
work> cd demo
demo> ls -p
CHANGELOG       app/            db/             log/            test/
README          components/     doc/            public/         vendor/
Rakefile        config/         lib/            script/
```

Tous ces dossiers (et les fichiers qu'ils contiennent) peuvent intimider au premier abord mais nous pouvons ignorer la majorité d'entre eux pour commencer. Dans ce chapitre nous n'en utiliserons que deux : le répertoire *app*, où nous écrirons notre application et le répertoire *script* qui contient quelques scripts très utiles.

Commençons par le répertoire *script*. L'un des scripts qu'il contient s'appelle `server`. Il lance un serveur Web isolé (également appelé autonome ou stand-alone) qui permet d'exécuter notre application nouvellement créée à l'aide de WEBrick (prononcez « Web-brique »).[1] Sans plus attendre, démarrons l'application que vous venez tout juste de créer.

```
demo> ruby script/server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
```

1. WEBrick est un serveur Web écrit en Ruby pur, livré avec Ruby 1.8.1 et ultérieur. Comme nous sommes sûrs qu'il est toujours présent, Rails l'utilise comme serveur Web de développement. Toutefois, si des serveurs Web comme Mongrel ou Lighttpd sont installés sur votre système (et que Rails peut trouver l'un d'entre eux) la commande **`script/server`** utilisera l'un d'eux plutôt que WEBrick. Vous pouvez forcer Rails à utiliser WEBrick en fournissant l'option suivante sur la ligne de commande :
`demo> ruby script/server <code:bold>webrick</code:bold>`

```
=> Ctrl-C to shutdown server; call with --help for options
 [2006-01-08 21:44:10] INFO  WEBrick 1.3.1
 [2006-01-08 21:44:10] INFO  ruby 1.8.2 (2004-12-30) [powerpc-darwin8.2.0]
 [2006-01-08 21:44:11] INFO  WEBrick::HTTPServer#start: pid=10138 port=3000
```

Comme l'indique la dernière ligne de la trace affichée au démarrage, nous venons juste de démarrer un serveur Web sur le port 3000 de notre machine[1]. Nous pouvons accéder à notre application Rails en pointant notre navigateur Web sur http://localhost:3000. Le résultat obtenu est représenté sur la figure 4-1.

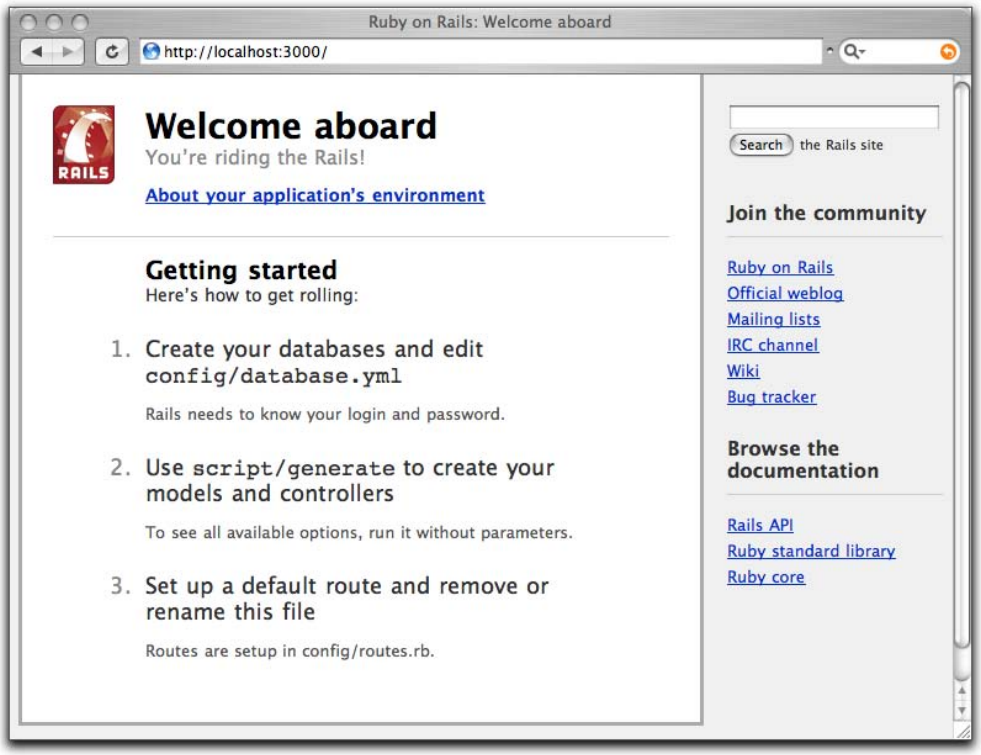


Figure 4-1
Application Rails nouvellement créée

1. L'adresse IP 0.0.0.0 présente dans l'URL signifie que WEBrick accepte les connections provenant de n'importe quelle interface réseau. Sur le système OS X de Dave, il s'agit à la fois de la connexion locale (127.0.0.1 et ::1) et de la connexion au réseau local (LAN).

Si vous regardez la fenêtre où vous avez démarré WEBrick, vous verrez la trace de vos accès à l'application. Nous allons laisser WEBrick tourner dans cette fenêtre shell. Plus tard, lorsque nous écrirons le code de notre application et que nous l'exécuterons, nous pourrons utiliser cette console pour tracer les requêtes entrantes. Lorsque vous n'avez plus besoin de votre application, vous pouvez taper `control-C` pour arrêter WEBrick (ne le faites pas tout de suite ; nous allons l'utiliser pour notre application dans une minute).

À ce stade, nous disposons d'une nouvelle application Rails opérationnelle où l'on ne trouve aucune trace de notre code. Nous allons y remédier sur-le-champ.

Bonjour, Rails !

Je n'y peux rien, il faut toujours que j'écrive un programme *Salut, tout le monde !* (*Hello, World !* en anglais) à chaque fois que j'utilise un nouveau système. L'équivalent en Rails serait une application qui envoie notre meilleur souvenir au navigateur.

Comme nous l'avons vu au chapitre 2, Rails est un framework de type modèle-vue-contrôleur. Il récupère les requêtes provenant d'un navigateur, les décode pour trouver un contrôleur et appelle une méthode d'action de ce contrôleur. Ensuite, le contrôleur invoque une vue particulière pour renvoyer le résultat à l'utilisateur. La bonne nouvelle c'est que Rails s'occupe de toute la plomberie interne qui lie toutes ces choses entre elles. Pour écrire notre application toute simple *Salut, tout le monde !*, nous devons produire un contrôleur et une vue. Nous n'avons pas besoin de coder un modèle pour le moment, puisque nous n'avons affaire à aucune donnée. Commençons par le contrôleur.

De façon analogue à la commande `rails` que nous avons employée pour créer une nouvelle application Rails, nous pouvons aussi utiliser un script et créer un nouveau contrôleur pour notre projet. Cette commande s'appelle `generate`, et elle est située dans le répertoire *`script`* du projet `demo` que nous venons de créer. Ainsi, pour créer un contrôleur nommé `Say`, assurons-nous d'être dans le répertoire *`demo`* puis exécutons le script en passant en paramètre le nom du contrôleur à créer[1].

```
demo> ruby script/generate controller Say
exists  app/controllers/
exists  app/helpers/
create  app/views/say
exists  test/functional/
create  app/controllers/say_controller.rb
create  test/functional/say_controller_test.rb
create  app/helpers/say_helper.rb
```

1. Le concept de « nom de contrôleur » est en fait plus complexe que vous ne le pensez. Nous l'expliquerons en détail dans le chapitre 14, section *Conventions de nommage*. Pour le moment, supposons simplement que le contrôleur s'appelle **`Say`**.

Le script affiche les noms des fichiers et des dossiers qu'il examine en indiquant chaque nouvelle création de fichier nécessaire à l'application. Dans l'immédiat, examinons l'un de ces scripts.

Le fichier source que nous allons regarder de près est le contrôleur. Vous le trouverez dans le fichier *app/controllers/say_controller.rb*. Examinons la définition des classes.

code/work/demo1/app/controllers/say_controller.rb

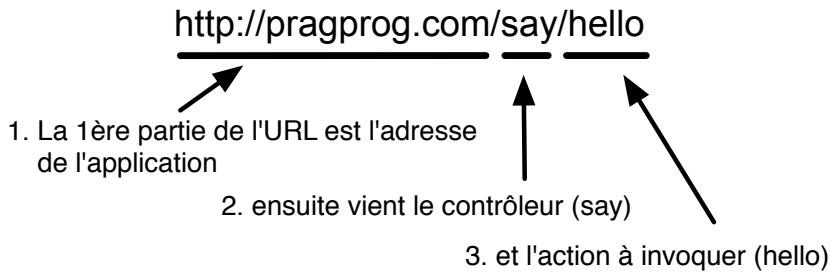
```
class SayController < ApplicationController
end
```

Plutôt minimaliste, n'est-ce pas ? `SayController` est une classe vide qui hérite de la classe `ApplicationController`. Ainsi reçoit-elle automatiquement tous les comportements du contrôleur par défaut de Rails. Ajoutons-y un peu de code pour que notre contrôleur puisse prendre en compte les requêtes entrantes. Que doit faire ce code ? Pour le moment rien du tout. Nous avons simplement besoin d'une méthode d'action vide. Donc la question suivante est : comment baptiser cette méthode ? Pour y répondre, nous devons comprendre comment Rails gère les requêtes.

Rails et les URL de requêtes

Comme toute autre application Web, une application Rails apparaît à ses utilisateurs comme étant associée à un pointeur Web aussi appelé URL (pour *Universal Resource Locator*). Quand vous pointez votre navigateur vers cette URL, vous appelez le code de l'application qui génère une réponse en retour.

Figure 4-2
Les URL sont mises en correspondance avec les contrôleurs et les actions



Cependant, le véritable processus est un peu plus complexe que cela. Imaginons que votre application soit accessible depuis l'URL http://pragprog.com/online/demo. Le serveur Web qui héberge votre application est plutôt malin quant à l'interprétation des chemins utilisés dans les URL. Il sait que les requêtes entrantes sur cette URL doivent dialoguer avec l'application. Quoiqu'on trouve à la suite de cette URL, la même application sera toujours invoquée. Tout morceau de chemin additionnel sera passé à l'application pour ses propres besoins.

Rails utilise le chemin pour déterminer le nom du contrôleur à utiliser et le nom de l'action à invoquer dans ce contrôleur[1]. Ceci est illustré par la figure 4-2. La première partie du chemin qui suit l'application est le nom du contrôleur et la seconde partie est le nom de l'action. C'est ce que représente la figure 4-3.

Notre première action

Ajoutons une action appelée `hello` à notre contrôleur `Say`. D'après ce qui a été dit dans la section précédente, nous savons que l'ajout de l'action `hello` nécessite la création d'une méthode `hello` dans la classe `SayController`. Mais que doit-elle faire ? Pour le moment, pas grand-chose. Souvenez-vous que le travail d'un contrôleur est de tout configurer correctement de façon à ce que la vue sache quoi afficher. Dans notre première application, il n'y a rien à configurer et une action vide suffira. Utilisez votre éditeur de texte favori pour modifier le fichier *`say_controller.rb`* dans le dossier *`app/controllers`*, en ajoutant la méthode `hello()` comme indiqué sur la figure ci-après.

code/work/demo1/app/controllers/say_controller.rb

```
class SayController < ApplicationController
  def hello
  end
end
```

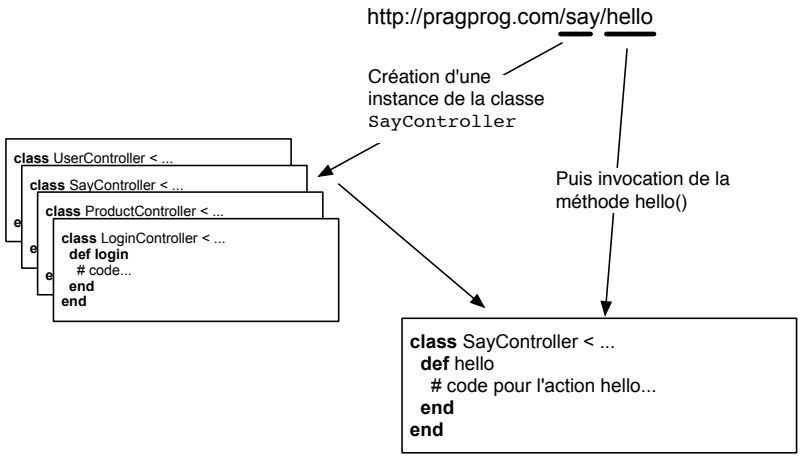
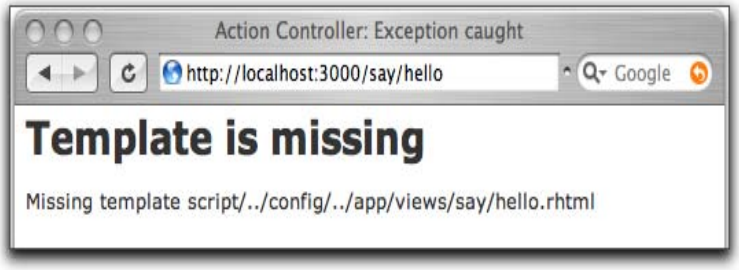


Figure 4-3
Les routes de Rails vers les contrôleurs et les actions.

1. Rails est plutôt souple quant au découpage et à l'analyse des URL entrantes. Dans ce chapitre, nous décrivons le mécanisme par défaut. Nous montrerons comment le redéfinir au chapitre 20, section *Routage des requêtes*.

Maintenant essayons d'appeler cette méthode. Ouvrez une fenêtre de votre navigateur et pointez-la sur l'URL http://localhost:3000/say/hello (notez que dans cet environnement de développement la première partie du chemin qui identifie habituellement l'application n'existe pas : nous routons directement les requêtes vers le contrôleur). En retour, vous devriez voir quelque chose qui ressemble à ce qui suit sur votre navigateur :



Cela peut paraître ennuyeux, mais l'erreur est parfaitement normale (sauf le chemin bizarre). Nous avons créé la classe contrôleur et la méthode d'action mais nous n'avons pas dit à Rails quoi afficher. Et c'est là que les vues entrent en jeu. Vous rappelez-vous de l'exécution du script qui a permis de créer le nouveau contrôleur ? La commande avait ajouté trois nouveaux fichiers et un répertoire à notre application. Ce répertoire contient des fichiers de formatage (que nous appellerons formats par la suite) pour les vues du contrôleur. Dans notre cas, nous avions créé un contrôleur appelé `Say` et les vues se trouvent donc dans le répertoire *`app/views/say`*.

Pour finir notre application *Salut, tout le monde !,* créons un format. Par défaut, Rails recherche les formats dans un fichier portant le même nom que l'action qu'il est en train d'exécuter. Dans notre cas, cela signifie que nous devons créer un fichier appelé *`hello.rhtml`* dans le répertoire *`app/views/say`* (pourquoi l'extension `.rhtml` ? Nous l'expliquerons dans un moment). Écrivons simplement quelques lignes HTML dans ce fichier.

code/work/demo1/app/views/say/hello.rhtml

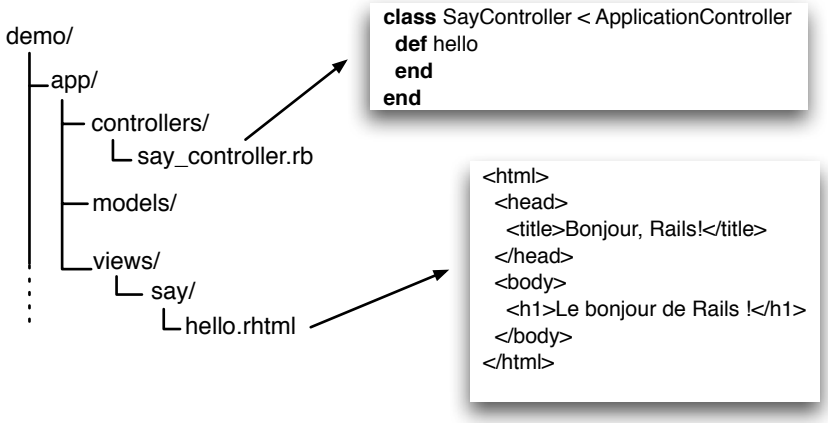
```
<html>
  <head>
    <title> Bonjour, Rails !</title>
  </head>
  <body>
    <h1> Le bonjour de Rails !</h1>
  </body>
</html>
```

Sauvegardez le fichier *`hello.rhtml`* et rechargez la page de votre navigateur Web. Maintenant vous devriez voir notre amicale salutation. Remarquez que nous n'avons pas eu à redémarrer

l'application pour observer les résultats de notre modification. En phase de développement, Rails prend automatiquement en compte les changements dès la sauvegarde des fichiers.

Jusqu'à présent, nous avons ajouté du code dans deux fichiers de l'arborescence de notre application Rails. Nous avons ajouté une action au contrôleur et nous avons créé un format pour afficher une page dans le navigateur. Ces fichiers résident à des emplacements standards de l'arborescence Rails : les contrôleurs dans `app/controllers`, et les vues dans `app/views`. Ceci est illustré par la figure 4-4.

Figure 4-4
Emplacements standards des contrôleurs et des vues



Soyons dynamiques

Jusque-là notre application est plutôt monotone. Elle n'affiche qu'une page statique. Pour la rendre un peu plus dynamique, ajoutons à la page l'affichage de l'heure courante.

Pour ce faire, nous devons modifier le fichier de formatage de la vue. Il doit maintenant inclure l'heure et la date courantes comme une chaîne de caractères. Cela soulève deux questions. Tout d'abord, comment ajoutons-nous du contenu dynamique dans un format ? Deuxièmement, comment récupérer l'heure et la date courantes ?

Contenu dynamique

Il existe deux façons de créer des formats dynamiques en Rails[1]. La première utilise une technologie appelée `Builder` que nous aborderons dans le chapitre 22, section *Formats de type Builder*. La seconde, que nous utiliserons ici, intègre du code Ruby dans le format lui-même. C'est pour cette raison que nous avons appelé le fichier de formatage *`hello.rhtml`* : l'extension `.rhtml` indique à Rails d'interpréter le contenu du fichier en utilisant un système appelé ERb (pour *Embedded Ruby*).

ERb est un filtre qui accepte, en entrée, un fichier avec une extension `.rhtml` et produit une version transformée en sortie. En Rails, la sortie en question est souvent du HTML mais cela pourrait être n'importe quel autre format. Du contenu normal passe à travers le filtre sans subir de modifications, alors que le contenu enserré par les balises `<%=` et `%>` est interprété comme du code Ruby et exécuté. Le résultat de cette exécution est converti en une chaîne de caractères et cette chaîne se substitue dans le format de départ à la séquence de code balisée par `<%=...%>`. Modifiez le fichier *`hello.rhtml`* de la façon suivante :

code/erb/ex1.rhtml

```
<ul>
  <li>Addition: <%= 1+2 %> </li>
  <li>Concaténation: <%= "cow" + "boy" %> </li>
  <li>L'heure dans une heure:  <%= 1.hour.from_now  %> </li>
</ul>
```

Quand vous rafraîchissez votre navigateur, le format génère maintenant le code HTML suivant :

code/erb/ex1.op

```
<ul>
  <li>Addition: 3 </li>
  <li>Concaténation: cowboy </li>
  <li>L'heure dans une heure:  Tue May 16 08:55:14 CDT 2006 </li>
</ul>
```

Dans la fenêtre de votre navigateur vous devriez voir quelque chose comme ce qui suit :

- `Addition : 3`
- `Concaténation : cowboy`
- `L'heure dans une heure : Sun May 07 16:06:43 CDT 2006`

1. En fait, il en existe trois, mais le troisième, rjs, est uniquement utilisé pour ajouter un peu de magie Ajax aux pages déjà affichées. Nous parlerons de rjs au chapitre 23, section *Formats RJS*.

De plus, ce qui se trouve entre `<%` et `%>` (sans signe égal) dans un fichier *`rhtml`* est exécuté comme du code Ruby mais sans substitution dans le fichier de sortie. Ce qui est néanmoins intéressant dans ce second type de traitement, c'est qu'on peut le mélanger avec du code autre que Ruby. Par exemple, nous pourrions écrire une version festive de notre fichier *`hello.rhtml`*.

code/erb/ex2.rhtml

```
<% 3.times do %>
Ho!<br />
<% end %>
Joyeux Noël!
```

Ce qui génère le HTML qui suit :

code/erb/ex2.op

```
Ho!<br />
Ho!<br />
Ho!<br />
Joyeux Noël!
```

Remarquez comme le texte du fichier inclus dans la boucle Ruby est envoyé vers la sortie à chaque itération.

Mais il se passe également quelque chose d'étrange. D'où viennent toutes ces lignes vides dans le HTML généré ? Et bien du fichier d'entrée. Si vous y réfléchissez bien, le fichier original contient un caractère de fin de ligne (ou plusieurs) tout de suite après la séquence `%>` de la première et de la troisième ligne du fichier. Conclusion : la séquence `<% 3.times do%>` est enlevée du fichier mais le caractère de fin de ligne reste. À chaque fois que la boucle est exécutée, ce caractère de nouvelle ligne est ajouté au fichier de sortie avec le texte complet de la ligne, y compris la ligne vide avant chaque *Ho*!. De la même manière, les caractères de nouvelle ligne après `<%` et `%>` sont responsables des lignes vides entre le dernier *Ho!* et la ligne *Joyeux Noël!*

En temps normal tout ceci n'a guère d'importance car le langage HTML ne prête aucune attention aux espaces et aux lignes vides. Mais si vous utilisez ce système de formatage pour créer des e-mails ou du HTML dans des blocs `<pre>`, il faut que vous supprimiez ces lignes vides. Pour cela, il suffit de changer la fin de la séquence ERb de `%>` en `-%>`. Ce signe moins signifie à Rails d'enlever tous les caractères nouvelle ligne (newline) du fichier de sortie. Si nous ajoutons un caractère moins sur la ligne `3.times`

code/erb/ex2a.rhtml

```
<% 3.times do -%>
Ho!<br />
<% end %>
Joyeux Noël!
```

nous obtenons le HTML suivant :

code/erb/ex2a.op

```
Ho!<br />
Ho!<br />
Ho!<br />
Joyeux Noël!
```

Et en ajoutant un moins sur la ligne `end`

code/erb/ex2b.rhtml

```
<% 3.times do -%>
Ho!<br />
<% end -%>
Joyeux Noël!
```

on supprime la ligne vide avant *Joyeux Noël*.

code/erb/ex2b.op

```
Ho!<br />
Ho!<br />
Ho!<br />
Joyeux Noël!
```

En général, supprimer ces lignes vides est une affaire de goût pas une nécessité. Cependant, il vous arrivera de lire du code Rails qui utilise le signe moins et il est donc préférable que vous sachiez quel est son rôle.

Dans l'exemple qui suit, la valeur affectée à la variable est intercalée dans le texte à chaque passage dans la boucle.

code/erb/ex3.rhtml

```
<% 3.downto(1) do |count| -%>
<%= count %>...<br />
<% end -%>
Décollage!
```

Voilà ce qui sera envoyé au navigateur :

code/erb/ex3.op

```
3...<br />
2...<br />
1...<br />
Décollage!
```

Faciliter le développement

Vous avez peut-être remarqué quelque chose concernant les développements que nous avons effectués jusqu'à présent. Au fur et à mesure que nous avons ajouté du code à notre application, nous n'avons pas eu à intervenir sur l'environnement d'exécution. Il tourne sans interruption en arrière-plan depuis le début de la session de travail, et pourtant, chaque changement effectué dans le code est immédiatement visible à chaque fois que nous accédons à l'application avec le navigateur. Comment est-ce possible ?

Il se trouve que le dispatcheur Rails basé sur WEBrick est plutôt astucieux. En mode développement (par opposition au mode de test ou de production), il recharge automatiquement les fichiers source de l'application à chaque nouvelle requête. De cette façon, lorsque nous modifions notre application, le dispatcheur est certain d'exécuter les modifications les plus récentes. C'est parfait pour le développement.

Toutefois, cette flexibilité a un coût. Elle engendre un court délai entre le moment où vous entrez l'URL et celui où l'application répond. Le travail de rechargement du dispatcheur en est la cause. En phase de développement, c'est un prix à payer qui en vaut la peine, mais en mode production, c'est inacceptable. C'est pourquoi cette fonctionnalité est désactivée lors du déploiement en production (voir chapitre 27).

Une dernière chose à propos de ERb. Il arrive que la valeur substituée par ERb lorsque vous utilisez `<%=...%>` comporte des caractères tels que inférieur à (`<`) ou et commercial (`&`) qui ont une signification en HTML. Pour éviter que votre page ne soit perturbée (et comme nous le verrons au chapitre 26, pour prévenir d'éventuels problèmes), il faut éviter l'emploi de ces caractères. Rails offre une méthode d'assistance (*helper* en anglais) appelée `h()` qui se charge de cela. Le plus souvent, il faut l'utiliser à chaque fois qu'une valeur est substituée dans les pages HTML.

code/erb/ex4.rhtml

```
Email: <%= h("Anne & Jacques <frazers@isp.email>") %>
```

Dans cet exemple, la méthode `h()` évite que les caractères spéciaux présents dans l'adresse e-mail n'interfèrent avec l'affichage de la page en les transformant en entités HTML. Le navigateur affiche effectivement *`Email: Anne & Jacques <frazers@isp.email>`*. Les caractères spéciaux sont affichés correctement.

Ajouter le temps

Notre problème initial était d'afficher la date et l'heure dans la page courante. Nous savons désormais comment afficher des données dynamiques dans notre application. Voyons maintenant comment obtenir la date et l'heure.

Une approche possible consiste à insérer un appel à la méthode Ruby `Time.now()` dans notre format *`hello.rhtml`*.

```
<html>
  <head>
    <title> Bonjour, Rails !</title>
  </head>
  <body>
    <h1>Le bonjour de Rails !</h1>
    <p>
      Il est <%= Time.now %>
    </p>
  </body>
</html>
```

Cela fonctionne. À chaque fois que nous accéderons à cette page, l'utilisateur verra l'heure courante substituée dans le corps de la réponse HMTL. Et pour notre application toute simple, cela peut être suffisant. En général, il est toutefois préférable de procéder de façon légèrement différente. Nous allons déplacer le calcul de l'heure à afficher dans le contrôleur et laisser à la vue le soin de l'afficher. Nous allons modifier notre méthode dans le contrôleur et positionner une variable d'instance appelée `@time` avec l'heure courante.

code/work/demo2/app/controllers/say_controller.rb

```
class SayController < ApplicationController
  def hello
    @time = Time.now
  end
end
```

Dans le format `rhtml` nous utilisons cette variable d'instance pour substituer l'heure dans le code HTML généré en sortie.

code/work/demo2/app/views/say/hello.rhtml

```
<html>
  <head>
    <title> Bonjour, Rails !</title>
  </head>
  <body>
    <h1> Le bonjour de Rails !</h1>
      <p>
        Il est <%= @time %>
      </p>
  </body>
</html>
```

Quand nous rafraîchissons notre fenêtre de navigation, nous voyons l'heure s'afficher au format standard Ruby comme indiqué sur la figure 4-5.

Figure 4-5
Salut, tout le monde ! Avec affichage de l'heure

Notez que l'heure change effectivement à chaque rechargement. Pas de doute, nous générons bien du contenu dynamique !

Pourquoi avoir pris la peine de positionner une variable avec l'heure à afficher dans le contrôleur pour ensuite l'utiliser dans la vue ? Bonne question. Dans cette application, vous pourriez tout simplement insérer l'appel `Time.now()` dans le format, mais en le plaçant dans le contrôleur vous vous octroyez quelques avantages. Par exemple, si nous souhaitons étendre notre application dans un futur proche pour qu'elle supporte des utilisateurs de différents pays, il nous faudra localiser l'affichage de l'heure en choisissant à la fois un format et un fuseau horaire appropriés. Tout cela finirait par donner pas mal de code de niveau applicatif qu'il ne serait pas du tout opportun de placer dans la vue. En positionnant l'heure à afficher dans le contrôleur, nous rendons notre application plus flexible : nous pouvons changer le format de

l'heure et le fuseau horaire dans le contrôleur sans avoir à mettre à jour aucune des vues qui utilisent cet objet temps. Le temps est ici une *donnée* et c'est donc le contrôleur qui doit la fournir à la vue. Nous verrons d'autres exemples similaires quand nous introduirons les modèles dans l'équation.

Comment la vue a-t-elle accès à l'heure ?

Dans la description des vues et des contrôleurs, nous avons montré que le contrôleur stocke l'heure courante dans une variable d'instance. Le format `rhtml` utilise ensuite cette variable d'instance pour substituer l'heure courante. Mais les données d'instance de cet objet contrôleur sont normalement accessibles à ce seul objet. En Ruby, on dit qu'elles sont privées (*private*). Comment se fait-il que ERb parvienne à accéder à ces données depuis le format ?

La réponse est à la fois simple et subtile. Rails a recours à un peu de magie Ruby de façon à injecter les variables d'instance de l'objet contrôleur dans l'objet format. En conséquence, le format de la vue peut accéder à n'importe quelle variable d'instance déclarée dans le contrôleur comme si elle étai la siennes

Certaines personnes insisteront sur ce point : mais comment ces variables sont-elles positionnées exactement ? Ces gens-là ne croient clairement pas à la magie. Évitez de passer Noël avec eux.

Résumé de l'histoire jusqu'à présent

Passons brièvement en revue la façon dont notre application fonctionne.

1. L'utilisateur se rend sur notre application. Dans notre cas, cela consiste à accéder à une URL telle que http://localhost:3000/say/hello.
2. Rails analyse l'URL. La partie `Say` est extraite comme nom du contrôleur, et Rails engendre donc une nouvelle instance de la classe Ruby `SayController` qu'il trouve dans le fichier *app/controllers/say_controller.rb*.
3. La partie suivante de l'URL, `hello`, identifie l'action. Rails invoque la méthode portant ce nom dans le contrôleur. Cette méthode d'action crée un nouvel objet `Time` qui contient l'heure courante, et le range dans la variable d'instance `@time`.
4. Rails cherche ensuite un format lui permettant d'afficher le résultat. Pour ce faire, il cherche un dossier portant le même nom que le contrôleur (*Say*) dans le répertoire *app/views*, et dans ce répertoire, un fichier portant le même nom que l'action (*hello.rhtml*).
5. Rails passe le format à travers ERb, en exécutant les instructions Ruby qui s'y trouvent et y substitue les valeurs positionnées par le contrôleur.
6. Le résultat est renvoyé au navigateur, et Rails termine le traitement de la requête.

Mais ce n'est pas tout. Rails vous donne de nombreuses occasions de modifier cet enchaînement par défaut et nous en tirerons parti très bientôt. Notre scénario, tel qu'il a été présenté, illustre parfaitement le fait que Rails s'appuie fortement sur des *conventions*. C'est véritable-

ment là un des piliers de la philosophie Rails. En proposant des comportements par défaut très pratiques et en appliquant certaines conventions, les applications Rails sont le plus souvent écrites avec peu ou pas du tout de fichier de configuration. Les choses s'articulent de façon naturelle.

Lier les pages entre elles

Il est rare qu'une application Web ne soit composée que d'une seule page. Voyons donc comment nous pouvons ajouter un autre exemple époustouflant de conception Web à notre application *Salut, tout le monde !*

Normalement, chaque type de page utilisé dans votre application va correspondre à une vue différente. Dans notre cas, nous allons aussi utiliser une nouvelle action pour gérer la page (bien que ce ne soit pas toujours le cas comme nous le verrons plus loin dans ce livre). Nous allons utiliser le même contrôleur pour les deux actions. De nouveau, ce n'est pas nécessairement le cas, mais nous n'avons aucune bonne raison de créer un nouveau contrôleur à ce stade.

Nous savons déjà comment ajouter une nouvelle vue et une nouvelle action à une application Rails. Pour ajouter l'action, nous définissons une nouvelle méthode dans le contrôleur. Appelons cette méthode `goodbye`. Notre contrôleur se présente maintenant ainsi :

code/work/demo3/app/controllers/say_controller.rb

```
class SayController < ApplicationController
  def hello
    @time = Time.now
  end

  def goodbye
  end
end
```

Ensuite nous devons créer un nouveau format dans le répertoire *app/views/say*. Cette fois, il s'appellera *goodbye.rhtml* puisque par défaut les fichiers de formatage portent le même nom que les actions associées.

code/work/demo3/app/views/say/goodbye.rhtml

```
<html>
  <head>
    <title>À plus tard !</title>
  </head>
  <body>
```

```
    <h1>Au revoir !</h1>
    <p>
      C'était un plaisir de vous voir.
    </p>
  </body>
</html>
```

Ouvrons maintenant notre fidèle navigateur et, cette fois, pointons-le vers notre nouvelle vue qui réside à l'URL http://localhost:3000/say/goodbye. Vous devriez voir quelque chose de similaire à la figure 4-6.

Figure 4-6
Un écran d'au revoir basique

Maintenant nous devons lier les deux écrans entre eux. Nous allons placer un lien dans l'écran de salutation qui nous amènera sur l'écran d'au revoir et vice versa. Dans une application réelle nous aurions sans doute recours à des boutons, mais pour le moment nous utiliserons simplement des hyperliens.

Nous savons déjà que Rails utilise une convention pour extraire de l'URL le contrôleur cible et une action dans ce contrôleur. Une approche simple consisterait donc à utiliser cette convention sur les URL pour nos hyperliens. Le fichier *hello.rhtml* contiendrait alors le code suivant :

```
<html>
  ...
  <p>
       Dit <a href="/say/goodbye">aurevoir</a>!
  </p>
   ...
```

et le fichier *goodbye.rhtml* pointerait dans le sens retour.

```
<html>
  ...
  <p>
```

```
    Dit <a href="/say/hello">bonjour</a>!
  </p>
  ...
```

Cette approche fonctionnerait sans aucun problème mais elle est trop fragile. Si nous décidions de déplacer notre application à un emplacement différent du serveur Web, les URL ne seraient plus valides. De plus, cela suppose que les URL suivent un format particulier qui pourrait très bien changer dans les futures versions de Rails.

Fort heureusement, nous n'avons pas besoin de prendre ces risques. Rails propose une collection de méthodes d'assistance (que nous appellerons désormais *assistants*) très utiles dans les formats. Dans le cas présent, nous utiliserons l'assistant `link_to()`, qui crée un hyperlien vers une action[1]. En utilisant `link_to()`, *`hello.rhtml`* devient :

code/work/demo4/app/views/say/hello.rhtml

```
<html>
  <head>
    <title> À plus tard !</title>
  </head>
  <body>
    <h1> Le bonjour de Rails !</h1>
    <p>
      Il est <%= @time %>.
    </p>
    <p>
      Il est temps de dire
      <%= link_to " Au revoir !", :action => "goodbye" %>
    </p>
  </body>
</html>
```

On y voit un appel à `link_to()` dans une séquence `<%=...%>` de ERb. Cet appel va créer un lien vers une URL qui invoquera l'action `goodbye()`. Le premier paramètre de l'appel à `link_to()` est le texte à afficher dans l'hyperlien et le paramètre suivant indique à Rails de générer un lien vers l'action `goodbye`. Comme nous ne spécifions pas le contrôleur de l'action cible, le contrôleur courant est utilisé.

Arrêtons-nous un instant sur la syntaxe du dernier paramètre de `link_to()`. Nous avons écrit :

```
link_to "Au revoir !", :action => "goodbye"
```

1. La méthode **`link_to()`** peut faire bien davantage, mais allons-y progressivement pour le moment...

Tout d'abord, `link_to` est un appel de méthode (en Rails, on appelle assistants les méthodes qui rendent l'écriture de formats plus facile). Si vous venez d'un langage tel que Java, vous serez peut-être surpris de constater l'absence de parenthèses autour des paramètres. Sachez que vous pouvez toujours en mettre si vous le souhaitez.

La partie `:action` dans le second paramètre est ce qu'on appelle un symbole Ruby. Vous pouvez voir le caractère deux points comme signifiant *la chose nommée...* Donc `:action` signifie *la chose nommée action*. Le `=> "goodbye"` associe la chaîne de caractères `goodbye` avec le nom `action`. En fait, cette forme syntaxique (qui n'est autre qu'un tableau associatif déguisé) permet d'utiliser ce qu'on appelle des paramètres nommés dans les appels de méthodes. Rails les utilise de façon intensive. Dès qu'une méthode possède un certain nombre de paramètres dont certains sont optionnels, vous pouvez utiliser ces paramètres nommés pour passer les valeurs de ces paramètres.

Revenons à notre application. Si nous pointons notre navigateur sur la page de salutation, elle contient maintenant un lien vers la page d'adieu, comme illustré sur la figure 4-7.

Figure 4-7
Page de salutation liée à la page d'adieu

Nous pouvons procéder au changement correspondant dans *`goodbye.rhtml`*, pour retourner vers la page de salutation.

code/work/demo4/app/views/say/goodbye.rhtml

```
<html>
  <head>
    <title> À plus tard !</title>
  </head>
  <body>
    <h1> Au revoir!</h1>
    <p>
      C'était un plaisir de vous voir.
    </p>
    <p>
```

```
      Dites <%= link_to " Bonjour", :action => "hello" %> à nouveau.
    </p>
  </body>
</html>
```

Ce que nous venons de faire

Au cours de ce chapitre nous avons construit une application excessivement simple. Au passage nous avons vu :

- comment créer une nouvelle application Rails et comment créer un contrôleur dans cette application ;
- comment Rails transforme les requêtes entrantes en appels dans votre code ;
- comment créer du contenu dynamique dans le contrôleur et l'afficher via un format de vue ;
- comment lier les pages entre elles.

Tout cela constitue une base solide qui va nous permettre de construire de vraies applications.

Récréation

Voici quelques petites choses à essayer par vous-même :

- Écrivez une page pour l'application *say* qui illustre l'utilisation des boucles qu'il est possible de construire avec ERb.
- Ajoutez puis supprimez le signe moins à la fin de la séquence ERb `<%= %>` (changez par exemple `%>` en `-%>` et vice versa). Utilisez l'option `Afficher > Source` pour constater la différence de formatage du HTML.
- L'appel à la méthode Ruby qui suit renvoie la liste de tous les fichiers du répertoire courant.

```
@files = Dir.glob('*')
```

- Utilisez-la pour positionner la valeur d'une variable d'instance dans une action du contrôleur puis écrivez le format correspondant qui permettra d'afficher les noms de fichiers dans une liste du navigateur.
- Indice : dans les exemples ERb, nous avons vu comment itérer n fois. Vous pouvez itérer sur une collection d'objets en utilisant quelque chose comme ceci :

```
<% for fichier in @fichiers %>
le nom du fichier est : <%= fichier %>
<% end %>
```

- Vous pouvez utiliser un élément `<ul>` pour construire la liste.

Vous trouverez d'autres indices sur http://wiki.pragprog.com/cgi-bin/wiki.cgi/RailsPlayTime.

Nettoyage

Vous avez peut-être lu ce chapitre en écrivant le code au fur et à mesure. Dans ce cas, il est fort probable que l'application tourne encore sur votre ordinateur. Lorsque nous commencerons à coder notre prochaine application, nous verrons un conflit apparaître lors de la première tentative d'exécution, car cette nouvelle application va tenter d'utiliser le même port 3000 que la première. Il est donc temps maintenant de stopper l'exécution de cette première application en tapant `control-C` dans la fenêtre où vous l'avez démarrée.