Ruby on Rails

Dave Thomas
David Heinemeier Hansson

ISBN : 2-212-11746-9.

EYROLLES

2

Architecture des applications Rails

L'une des particularités de Rails est d'imposer quelques contraintes assez sérieuses sur la façon de structurer vos applications Web. De façon assez étonnante, ces contraintes rendent la création d'applications beaucoup plus facile. Voyons pourquoi.

Modèles, vues et contrôleurs

En 1979, Trygve Reenskaug suggéra une nouvelle architecture pour le développement d'applications interactives. Selon ce design, les applications étaient divisées en 3 composants : modèles, vues et contrôleurs.

Le *modèle* est en charge de maintenir l'état de l'application. Parfois cet état est transitoire et ne dure que le temps de quelques interactions avec l'utilisateur. Parfois l'état est permanent et il est stocké hors de l'application, souvent dans une base de données.

Un modèle est plus qu'une collection de données ; il impose toutes les règles métier qui s'appliquent aux données. Par exemple, si un rabais ne doit pas être appliqué aux commandes d'un montant inférieur à 20 euros, c'est le modèle qui appliquera cette règle. Ceci est parfaitement sensé : en implémentant les règles métier dans le modèle, nous nous assurons que rien d'autre dans notre application ne peut rendre nos données invalides. Le modèle est à la fois le gardien et l'entrepôt de nos données.

La *vue* est en charge de la génération de l'interface utilisateur, normalement basée sur les données du modèle. Par exemple, une boutique en ligne dispose d'une liste de produits à afficher à l'écran sous forme d'un catalogue. Cette liste est accessible via le modèle, mais c'est la

vue qui accède à la liste depuis le modèle et qui la formate pour l'utilisateur final. Bien que la vue puisse proposer à l'utilisateur différents moyens de saisir des données, elle ne gère jamais les données saisies par elle-même. Le travail de la vue est bel et bien terminé une fois les données affichées. Il peut exister plusieurs vues répondant à différents besoins et qui accèdent au même modèle de données. Dans la boutique en ligne, il y aura par exemple une vue qui affiche les informations d'un produit pour la page du catalogue et un autre ensemble de vues visibles par les administrateurs pour ajouter ou modifier un produit.

Les *contrôleurs* orchestrent l'application. Les contrôleurs reçoivent les événements du monde extérieur (normalement saisis par l'utilisateur), interagissent avec le modèle et affichent ensuite une vue appropriée pour l'utilisateur.

Ce triumvirat – le modèle, la vue et le contrôleur – forme une architecture connue sous le nom de MVC. La figure 2-1 montre une vue synthétique d'une architecture MVC.

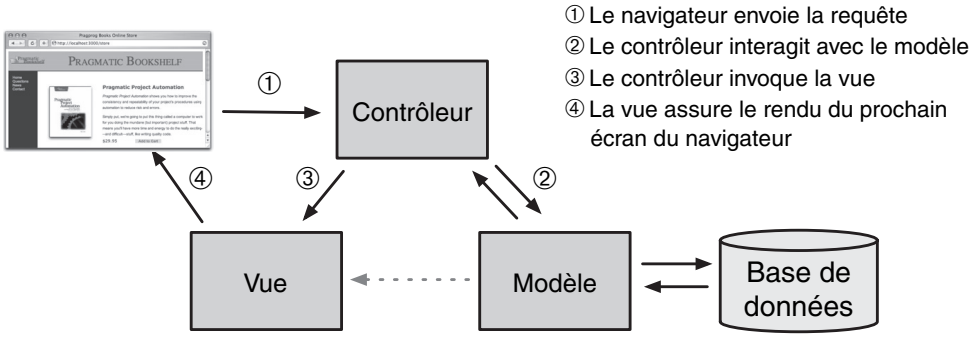

Figure 2-1
L'architecture modèle-vue-contrôleur

MVC a été initialement utilisée pour les applications graphiques conventionnelles. Les développeurs ont alors découvert que la séparation des problèmes réduisait le couplage, ce qui par voie de conséquence, donnait un code plus facile à écrire et à maintenir. Chaque concept ou action était exprimée à un seul endroit parfaitement connu. Utiliser MVC revenait à construire un gratte-ciel dont les poutrelles étaient déjà en place. Du même coup, il devenait beaucoup plus facile d'accrocher le reste des pièces à la structure.

Dans le monde du logiciel, nous ignorons souvent les bonnes idées du passé dans notre course effrénée vers le futur. Quand les développeurs ont commencé à produire des applications Web, ils sont revenus à l'écriture de programmes monolithiques qui mélangeaient allègrement présentation, accès à la base de données, logique métier et gestion des événements dans un seul gros morceau de code. Mais les idées du passé ont petit à petit refait surface, et les mêmes développeurs ont commencé à expérimenter des architectures d'applications Web qui reprenaient les idées du modèle MVC vieilles de 20 ans. C'est ainsi que sont nés des

frameworks tels que WebObjects, Struts, et JavaServer Faces. Tous sont basés (plus ou moins fidèlement) sur les idées de l'architecture MVC.

Ruby on Rails est également un framework MVC. Rails impose une structure à votre application faite de modèles, de vues et de contrôleurs, qui sont autant de fonctionnalités séparées que Rails assemblera au moment de l'exécution. L'un des plaisirs de Rails est que cet assemblage est fondé sur l'utilisation de comportements, par défaut intelligents, qui font que vous n'avez besoin d'écrire aucune métadonnée de configuration que ce soit pour faire fonctionner l'ensemble. C'est un exemple de la philosophie de Rails qui favorise la convention plutôt que la configuration.

Dans une application Rails, les requêtes entrantes sont en premier lieu envoyées à un routeur, qui détermine dans quelle partie de l'application la requête doit être aiguillée et comment elle doit être analysée. Au final, cette phase identifie une méthode particulière (appelée *action* dans le jargon Rails) située quelque part dans le code du contrôleur. L'action peut consulter les données qui accompagnent la requête elle-même, elle peut interagir avec le modèle et déclencher l'invocation d'autres actions. Enfin, l'action prépare les informations nécessaires à la vue chargée de présenter un résultat à l'utilisateur.

La figure 2-2 montre comment Rails traite une requête entrante. Dans cet exemple, imaginez que l'application vienne juste d'afficher une page d'un catalogue de produits à l'écran et que l'utilisateur ait cliqué sur le bouton `Ajouter au panier` situé à côté de l'un des produits. Ce bouton sollicite notre application en utilisant l'URL http://my.url/store/add_to_cart/123, où 123 est l'identifiant (id) interne du produit sélectionné.[1]

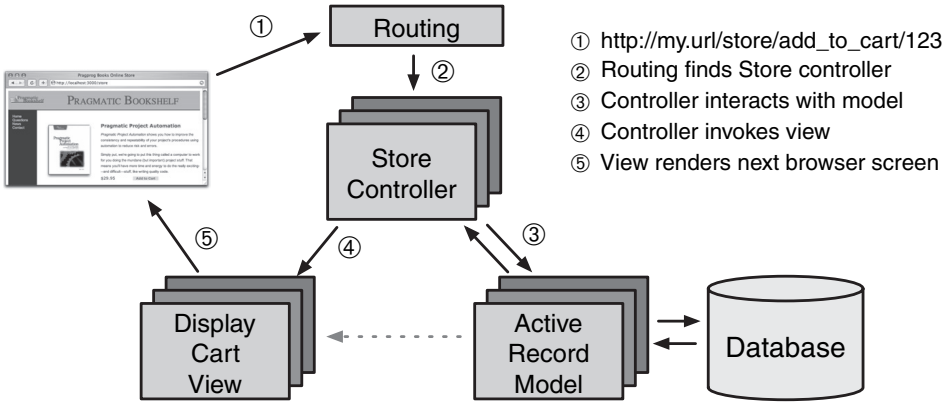

Figure 2-2
Rails et MVC

1. Nous traiterons du format des URL de Rails plus loin dans cet ouvrage. Néanmoins, il n'est pas inutile de mentionner ici qu'utiliser des URL entraînant des actions comme *Ajouter au panier* peut être dangereux. Voir chapitre 16, *Action Controller et Rails* pour plus de détails.

Le composant en charge du routage reçoit la requête entrante et la découpe immédiatement en morceaux. Dans cet exemple, il interprète la première partie du chemin, `store`, comme étant le nom d'un contrôleur et `add_to_cart`, comme étant le nom d'une action. La dernière partie du chemin, `123`, est par convention affectée à un paramètre interne appelé `id`. Après cette analyse, le routeur sait désormais qu'il doit invoquer la méthode `add_to_cart()` dans la classe du contrôleur `StoreController` (les conventions de nommage de Rails sont traitées page 202).

La méthode `add_to_cart()` traite la demande de l'utilisateur. Dans le cas présent elle se charge de trouver le panier de l'utilisateur courant (qui est un objet géré par le modèle). Elle demande aussi au modèle de trouver les informations concernant le produit 123. Ensuite elle demande à l'objet panier d'ajouter le produit à lui-même (au passage notez comment le modèle est utilisé pour garder la trace de toutes les données métier ; le contrôleur lui dit *quoi* faire et le modèle sait *comment* le faire).

Maintenant le panier contient un nouveau produit, nous pouvons donc le montrer à l'utilisateur. Le contrôleur s'arrange pour que la vue ait accès à l'objet panier depuis le modèle et invoque le code qui génère la vue. Dans Rails, cette invocation est souvent implicite ; une fois de plus les conventions adoptées par Rails aident à relier automatiquement une vue à une action donnée.

Et voilà véritablement ce qu'il y a à dire d'une application Web MVC. En se conformant à un ensemble de conventions et en découpant vos fonctionnalités de façon appropriée, vous découvrirez qu'il devient plus facile de travailler avec votre code et que votre application devient plus simple à étendre et à maintenir. Ça ressemble à une bonne affaire, n'est-ce pas ?

Si MVC se résume simplement à une façon de découper votre code, vous vous demandez peut-être pourquoi vous avez besoin d'un framework tel que Ruby on Rails. La réponse est assez simple : Rails se charge de toute la basse besogne pour vous – toute cette quantité de détails qui vous prend tellement de temps à gérer personnellement – et vous laisse vous concentrer sur les fonctionnalités qui sont le cœur de votre application. Voyons comment...

Active Record : le support du modèle Rails

En général, nos applications Web conservent leurs informations dans une base de données relationnelle. Les systèmes de saisie de commandes y stockent leurs commandes, les articles commandés et les détails à propos des clients dans les tables de la base de données. Même les applications qui utilisent normalement du texte non structuré, tels que les blogs et les sites d'annonces, utilisent souvent des bases de données comme entrepôt de données.

Bien que ça ne vous ait peut-être pas sauté aux yeux lorsque vous utilisez SQL pour y accéder, les bases de données relationnelles sont en fait construites sur la théorie mathématique des ensembles. Bien que ce soit une bonne chose du point de vue conceptuel, cela complique le mariage des bases de données relationnelles avec les langages de programmation orientés objet. Les objets ne sont que données et opérations alors que les bases de données relationnelles ne gèrent que des ensembles de valeurs. Des choses simples à exprimer en termes rela-

tionnels sont parfois difficiles à coder dans des systèmes orientés objet. L'inverse est également vrai.

Au fil du temps, les programmeurs ont trouvé différentes façons de réconcilier les vues relationnelles et orientées objet de leur données d'entreprise. Jetons un coup d'œil à deux approches différentes. L'une organise les programmes autour de la base de données ; l'autre organise la base de données autour du programme.

Programmation centrée sur la base de données

Dans la première approche, les développeurs qui codaient sur une base de données relationnelle utilisaient généralement des langages comme C ou COBOL. Ils imbriquaient habituellement le SQL[1] directement dans leur code, soit sous forme de chaînes de caractères, soit en utilisant un préprocesseur qui convertissait le SQL présent dans le source en appel de plus bas niveau vers le moteur de base de données.

Avec une telle imbrication il devenait naturel d'entremêler la logique de la base de données avec la logique d'ensemble de l'application. Un développeur qui souhaitait parcourir la liste de commandes et mettre à jour le taux de TVA de chaque commande était amené à écrire quelque chose de très laid dans cette veine :

```
EXEC SQL BEGIN DECLARE SECTION;
  int   id;
  float amount;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c1 AS CURSOR FOR
select id, amount from orders;
while (1) {
  float tax;
  EXEC SQL WHENEVER NOT FOUND DO break;
  EXEC SQL FETCH c1 INTO :id, :amount;
   tax = calc_sales_tax(amount)
  EXEC SQL UPDATE orders set tax = :tax where id = :id;
}
EXEC SQL CLOSE c1;
EXEC SQL COMMIT WORK;
```

Effrayant non ? Ne vous inquiétez pas, nous ne ferons rien dans ce style bien que ce soit chose courante dans des langages de scripting comme Perl ou PHP. C'est aussi faisable en Ruby. Par exemple, nous pourrions utiliser le module DBI de Ruby pour produire du code ressemblant

1. SQL, abréviation de Structured Query Language, est le langage utilisé pour interroger et mettre à jour les bases de données relationnelles.

à celui qui précède (remarque : cet exemple, comme le précédent, ne comporte pas de contrôle d'erreur).

```
def update_sales_tax
  update = @db.prepare("update orders set tax=? where id=?")
  @db.select_all("select id, amount from orders") do |id, amount|
    tax = calc_sales_tax(amount)
    update.execute(tax, id)
  end
end
```

Cette approche est concise et simple et, de fait, elle est très utilisée. Cela semble une solution idéale pour de petites applications mais elle souffre en réalité d'un handicap majeur. Mélanger ainsi la logique métier avec les accès à la base de données rend la maintenance et l'extension du code difficile. De plus vous avez aussi besoin de connaître SQL dès que vous commencez à développer votre application.

Supposons par exemple que notre gouvernement, bien inspiré, fasse voter une nouvelle loi qui stipule que la date et l'heure auxquelles le taux de TVA a été calculé doit figurer dans la base de données. Pas de problème, pensons-nous. Nous n'avons qu'à demander l'heure courante dans notre boucle de contrôle, ajouter une colonne à l'instruction SQL `update` et passer la valeur de temps en paramètre lors de l'appel à `execute()`.

Mais que se passe-t-il si nous renseignons la colonne TVA à de nombreux endroits dans l'application ? Nous devrons alors parcourir le code à la recherche de toutes ces occurrences et mettre à jour chacune d'elles. Nous avons non seulement dupliqué du code mais, si par mégarde nous avons oublié de modifier un endroit du code où cette colonne est renseignée, nous avons de surcroît introduit une anomalie.

Les techniques de programmation orientées objet nous ont appris que l'encapsulation résout ce genre de problèmes. En regroupant tout ce qui a trait aux traitements de commandes dans une classe nous n'avons plus qu'un seul endroit à mettre à jour si la réglementation change.

Les programmeurs ont donc étendu ces idées à la programmation des bases de données. Le principe de base est trivial. Nous emballons les accès à la base de données dans une couche de classes. Le reste de l'application utilise ces classes et leurs objets sans jamais interagir directement avec la base de données. Dans le cas de notre changement concernant la TVA, nous modifierions simplement la classe qui enveloppe la table des commandes afin de mettre à jour le marqueur de temps dès que la TVA change.

En pratique ce concept est plus difficile à mettre en œuvre qu'il n'y paraît. Dans la vraie vie, les tables de bases de données sont reliées entre elles (une commande peut comporter plusieurs articles par exemple) et nous souhaiterions que nos objets reflètent cela. Autrement dit, l'objet commande devrait contenir une collection d'objets article. Et c'est précisément là que nous commençons à entrer dans des problèmes de navigation dans les objets, de perfor-

mance et de cohérence de données. Quand elle est confrontée à ce genre de complications, l'industrie informatique fait ce qu'elle a toujours fait : elle invente un acronyme à 3 lettres. En l'occurrence il s'agit de ORM pour Object/Relational Mapping. Comme vous l'aurez compris, Rails utilise ORM.

Correspondance objet/relation

Les bibliothèques ORM mettent en correspondance des tables de base de données avec des classes. Ainsi une table de la base de données appelée `orders` sera reliée à la classe nommée `Order`. Les lignes de cette table correspondent aux objets de la classe. Une commande particulière est représentée par un objet de type `Order`. Au sein de cet objet, des attributs sont utilisés pour lire ou écrire la valeur de chaque colonne. Notre objet `Order` possède des méthodes pour lire et écrire la valeur du montant, de la TVA, etc.

De plus, les classes Rails qui enveloppent les tables de notre base de données fournissent un ensemble de méthodes de classe qui exécutent des opérations au niveau des tables. Par exemple, nous pourrions avoir besoin de trouver une commande avec un identifiant (id) particulier et d'afficher ensuite le montant (attribut amount) de cette commande. Ceci est implémenté comme une méthode de classe qui retourne l'objet `Order` correspondant. En Rails, cela donne :

```
order = Order.find(1)
puts "Order #{order.client_id}, amount=#{order.amount}" #puts
```

Parfois ces méthodes de classe retournent non pas un mais une collection d'objets.

```
Order.find(:all, :conditions => "name='dave'") do |order| #
iteration
  puts order.amount
end
```

Enfin, les objets correspondants à chacune des lignes d'une table possèdent des méthodes qui opèrent sur cette ligne. La plus utilisée est probablement la méthode `save()`, une opération qui sauvegarde la ligne correspondant à un objet dans la base de données.

```
Order.find(:all, :conditions => "nom='dave'") do |order|
  order.discount = 0.5
  order.save
end
```

En résumé une couche ORM fait correspondre les tables à des classes, les lignes de la table aux objets et les colonnes de la table aux attributs de ces objets. Les méthodes de classe sont

utilisées pour exécuter des opérations au niveau de la table et les méthodes d'instance opèrent au niveau d'une ligne en particulier.

Dans une bibliothèque ORM typique, il faut fournir des paramètres de configuration pour spécifier la correspondance entre les entités de la base de données et les entités manipulées par votre programme. Les programmeurs qui utilisent ce genre d'outils ORM se retrouvent souvent à créer ou à maintenir un grand nombre de ces fichiers de configuration souvent écrits en XML.

Active Record

Active Record est la couche ORM de Rails. Active Record suit de près le modèle ORM standard : les tables correspondent aux classes, les lignes aux objets et les colonnes aux attributs de l'objet. Elle diffère de la plupart des autres bibliothèques ORM sur la façon dont elle se configure. En s'appuyant sur des conventions et en ajoutant des comportements par défaut sensés, Active Record réduit le temps nécessaire à la configuration. Pour illustrer ceci, voici un programme qui utilise Active Record pour envelopper notre table `orders`.

```
require 'active_record'
class Order < ActiveRecord::Base
end
order = Order.find(1)
order.discount = 0.5
order.save
```

Ce code utilise la nouvelle classe `Order` pour récupérer la commande dont l'identifiant est 1 et modifier le rabais associé (nous avons omis le code qui crée la connexion à la base de données pour le moment). Active Record nous épargne le souci d'avoir à traiter directement avec la base de données, nous laissant ainsi libre de nous concentrer sur les règles métier.

Mais Active Record fait plus que cela. Comme vous le verrez lorsque nous développerons notre application de boutique en ligne (à partir de la page 47), Active Record s'intègre de façon transparente avec le reste du framework Rails. Si un formulaire Web contient des données relatives à un objet métier, Active Record peut extraire automatiquement les données et les transformer en un objet du modèle correspondant. Active Record offre aussi une vérification sophistiquée du modèle de données. Ainsi, si les données d'un formulaire Web sont invalides, la vue Rails peut extraire et formater les erreurs d'une simple ligne de code.

Active Record est la pierre angulaire de l'architecture MVC de Rails capable de gérer tous nos modèles. Voilà pourquoi nous lui consacrons deux chapitres complets à partir de la page 213.

Action Pack : la vue et le contrôleur

En y réfléchissant un peu, les parties vue et contrôleur de MVC sont assez intimement liées. Le contrôleur fournit des données à la vue et le contrôleur reçoit en retour des événements provenant des pages générées par les vues. En raison de ces interactions fortes, le support des vues et des contrôleurs dans Rails est regroupé dans un seul composant appelé *Action Pack*.

N'en déduisez pas pour autant que le code des vues et des contrôleurs de votre application sera mélangé uniquement parce que Action Pack est un composant unique. C'est tout le contraire ; Rails vous donne le niveau de séparation requis pour l'écriture de vos applications en démarquant clairement le code dédié à la logique de contrôle de celui dévolu à la logique de présentation.

Support de la vue

En Rails, la vue est responsable de la création de la page à afficher dans un navigateur en tout ou partie[1]. Dans sa forme la plus simple, une vue est un morceau de code HTML qui affiche un texte fixe. Plus couramment, elle comprend du contenu généré dynamiquement par la méthode d'action du contrôleur.

En Rails, le contenu dynamique est généré par des fichiers de formatage (aussi appelés formats) qui existent en deux variantes. La première insère du code Ruby dans le code HTML de la vue en s'appuyant sur un outil Ruby appelé ERb (pour Embedded Ruby)[2]. Cette approche est très flexible mais les puristes lui reprochent parfois de violer l'esprit de l'architecture MVC. En intégrant du code dans la vue nous risquons en effet d'y introduire de la logique qui devrait se trouver soit dans le modèle soit dans le contrôleur. Ce reproche est largement infondé : les vues contenaient du code actif même dans les architecture MVC originelles. Maintenir une séparation claire entre les problèmes fait partie du travail d'un développeur (nous reviendrons sur les formats, ou template, HTML dans le chapitre 17, *Action View*).

Rails offre aussi une seconde variante qui définit les vues de type *Builder*. Cette seconde approche permet de construire des documents XML à partir de code Ruby. La structure du XML généré suivra automatiquement la structure du code. Nous couvrirons les modèles de style constructeur à partir de la page 363.

Et le contrôleur !

Le contrôleur Rails est le centre logique de votre application. Il coordonne les interactions entre l'utilisateur, les vues et le modèle. Cependant, Rails s'affranchit de cette tâche de façon

1. Il peut aussi s'agir d'une réponse XML, ou d'un e-mail, etc. Le point clé est que la vue génère la réponse qui sera envoyée à l'utilisateur quelle que soit sa forme.
2. Cette approche semblera familière aux programmeurs d'applications Web travaillant avec les technologies PHP ou Java JSP.

quasi transparente pour le programmeur ; le code que vous écrivez n'a donc à se préoccuper que des fonctionnalités de l'application. Ceci rend le code des contrôleurs Rails particulièrement simple à développer et à maintenir.

Le contrôleur abrite aussi un nombre important de services auxiliaires.

- Il est chargé de router les requêtes externes vers les actions internes. Il gère parfaitement bien les URL qui sont faciles à lire pour l'utilisateur.
- Il gère le cache, ce qui peut amener une amélioration du niveau de performance de plusieurs ordres de grandeur pour vos applications.
- Il gère les modules d'aide qui étendent les capacités des formats utilisés dans les vues sans encombrer leur code.
- Il gère les sessions, donnant ainsi l'impression aux utilisateurs d'une interaction continue avec nos applications.

Le framework Rails est d'une grande richesse. Et plutôt que de nous engager dans une revue détaillée composant par composant, nous allons retrousser nos manches et écrire de véritables petites applications. Dans le chapitre qui suit nous allons d'abord installer Rails. Ensuite nous écrirons quelque chose de simple pour nous assurer que tout est installé correctement. Dans le chapitre 5, *Présentation de l'étude de cas*, nous commencerons à écrire quelque chose de plus conséquent avec une application de boutique en ligne.