Ruby on Rails

Dave Thomas
David Heinemeier Hansson

ISBN : 2-212-11746-9.

EYROLLES

18

Le Web 2.0 avec AJAX

Ce chapitre a été écrit par Thomas Fuchs (http://mir.aculo.us/), architecte logiciel qui réside à Vienne, Autriche. Il travaille sur des applications Web depuis 1996. Il est à l'origine de plusieurs techniques décrites dans ce chapitre : les champs de formulaire à complétion (complément) automatique, la plupart des effets visuels et l'indicateur de progression du téléchargement de fichier.

Deux choses ont pénalisés les développeurs d'applications depuis que le Web est né :

- l'absence d'état des connexions HTTP ;
- l'impossibilité de solliciter le serveur entre l'affichage de deux pages successives.

L'absence d'état dans le protocole HHTP a rapidement été comblée par l'arrivée des cookies qui permettent d'identifier l'origine des requêtes et de stocker une session du côté du serveur. C'est à cet effet que Rails propose l'objet `session`.

Le second problème n'était pas si facile à résoudre. L'introduction des balises HTML `<frameset>` et `<frame>` a longtemps représenté une solution partielle à ce problème mais les inconvénients étaient tels que les développeurs Web qui utilisaient ces balises finissaient tous, un jour ou l'autre, par se taper la tête contre les murs. Quelqu'un eut alors l'idée d'inventer la balise `<iframe>`, mais cela n'a pas beaucoup arrangé la situation.

À une époque où les interfaces utilisateurs accélérées à grand coup d'Open-GL font la loi sur les machines de bureau, la plupart des applications Web semblent tout droit sorties d'un terminal de saisie des années 1960.

Tout cela fait maintenant parti du passé.

Bienvenue sur le Web, version 2.0.

Voici AJAX

AJAX (acronyme de Asynchronous JavaScript and XML)[1] est une technique qui étend les applications Web en leur permettant de lancer des requêtes vers le serveur depuis une page du navigateur.

Qu'est-ce que cela signifie en pratique ? Cela permet d'effectuer certaines choses, du même type que ce que vous pouvez voir si vous utilisez GMail, Google Maps ou Google Suggest. Certaines pages se comportent exactement comme une application de bureau classique. Comment est-ce possible ? Normalement le serveur envoie une page, vous cliquez sur quelque chose qui déclenche une autre requête vers le serveur, qui lui-même répond par une nouvelle page.

Avec AJAX, tout change. Vous pouvez déclencher à la volée des requêtes depuis le client Web vers le serveur et le serveur peut répondre en renvoyant toutes sortes de choses très utiles :

- des fragments HTML ;
- des scripts à exécuter côté client ;
- des données quelconques.

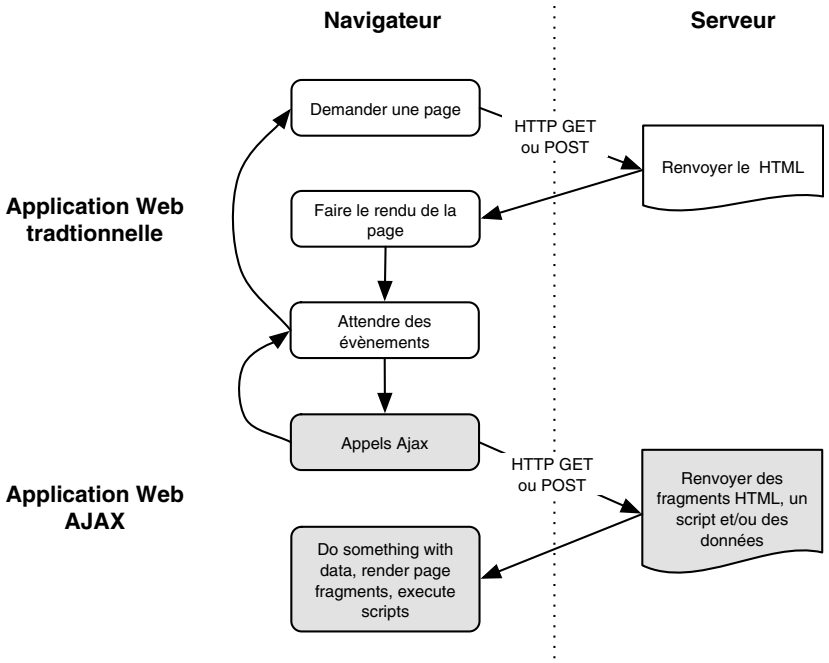


Figure 18-1
AJAX – requêtes internes à la page pour les applications Web

1. Le terme fut inventé par la société Adaptive Path. Pour plus d'informations voir l'essai de James Garrett sur http://www.adaptivepath.com/publications/essays/archives/000385.php.

En retournant des *fragments HTML* depuis le serveur, AJAX nous permet de remplacer ou d'ajouter des éléments dans la page HTML affichée sans avoir à la recharger en intégralité. Vous pouvez ainsi remplacer ici un paragraphe, là l'image d'un produit ou bien une ligne dans une table. Non seulement ce genre d'interaction entre le client et le serveur réduit l'utilisation de la bande passante mais les pages réagissent plus rapidement aux sollicitations de l'utilisateur.

En exécutant des *scripts* (JavaScript bien sûr) renvoyés par le serveur, il est possible de changer complètement l'aspect, le contenu et le comportement de la page affichée.

Enfin, il est possible de renvoyer des *données quelconques* qui seront traitées par un script côté client[1].

XMLHttpRequest

AJAX s'appuie sur une fonctionnalité du navigateur que Microsoft fût le premier à mettre en œuvre dans Internet Explorer, mais qui depuis, s'est propagée aux autres navigateurs (ceux basés sur Gecko comme Firefox ou Mozilla et d'autres comme Opera ou Safari d'Apple). Cette fonctionnalité est matérialisée par un objet JavaScript *XMLHttpRequest*.

Cet objet permet non seulement d'initier des requêtes HTTP du client vers le serveur mais aussi de traiter les données reçues en retour.

À noter que le préfixe *XML* de la classe `XMLHttpRequest` se trouve là pour des raisons historiques (eh oui, même les technologies les plus récentes ont une histoire !) et vous n'êtes en aucun cas tenus d'utiliser XML pour le format de vos données. Oubliez donc le XML pour le moment et prenez cet objet pour ce qu'il est : un habillage autour d'une requête HTTP.

Le A de AJAX

Les appels AJAX sont asynchrones, ou *non bloquants* si vous préférez[2]. Après l'envoi de la requête vers le serveur, la boucle d'événements principale reprend son travail en surveillant à la fois les événements classiques de type clic de souris mais aussi les réponses reçues par l'objet `XMLHttpRequest`. Cela signifie que les données AJAX retournées par le serveur constituent juste un événement comme les autres. Entre l'envoi de la requête et le retour de la réponse tout peut arriver. Gardez bien cela en tête lorsque votre application commencera à devenir de plus en plus complexe.

1. On s'attend bien sûr à ce que la partie *XML* d'AJAX entre en jeu à cet endroit là, car les serveurs sont censés renvoyer des données au format XML vers le client. Mais sur le fond rien ne nous y oblige. Vous pouvez renvoyer des fragments de code JavaScript à exécuter, du texte ou même du YAML.
2. Il est aussi possible de faire des appels synchrones mais c'est une très, très mauvaise idée car votre navigateur cessera de répondre à toutes vos sollicitations jusqu'à ce que la requête ait été traitée.

XMLHttpRequest contre <iframe>

Pourquoi donc cet engouement pour AJAX ? Je fais ça depuis des années avec la balise `<iframe>` ! et tout se passe très bien.

Bien qu'il soit tout à fait vrai que les deux paraissent assez semblables de prime abord, les iframes ne sont ni aussi propres ni aussi flexibles qu'AJAX. Contrairement à l'approche `<iframe>`, avec AJAX :

- il est facile d'émettre toutes sortes de requêtes (GET, POST, etc.) ;
- le DOM (Document Objet Model) n'est pas du tout altéré ;
- il existe des procédures de rappel (callbacks) puissantes ;
- l'interface de programmation est propre ;
- il possible de personnaliser les en-têtes HTTP.

En tenant compte de tout cela il paraît clair que `XMLHttpRequest` offre un modèle de programmation bien plus puissant et bien plus propre que celui des iframes.

AJAX à la mode Rails

Le support des appels AJAX est intégré dans Rails, ce qui vous permet d'amener très facilement votre application au niveau du Web, version 2.0.

Tout d'abord l'implémentation AJAX de Rails propose en standard les bibliothèques JavaScript *prototype*[1], *effects*, *dragdrop* et *controls*. Ces bibliothèques encapsulent toutes sortes d'opérations de manipulation AJAX et DOM dans une approche orientée objet très élégante.

D'autre part nous disposons aussi de *JavascriptHelper*, module qui définit une série de méthodes que nous détaillerons dans la suite du chapitre. Ces méthodes habillent la mécanique JavaScript avec une façade Ruby de façon à ce que vous n'ayez pas à basculer constamment entre les deux langages de programmation lorsque vous utilisez AJAX. C'est ce qui s'appelle de l'intégration totale.

Afin d'utiliser les fonctions de `JavascriptHelper`, il faut d'abord inclure le fichier *`prototype.js`* dans l'application en utilisant l'appel suivant dans la section `<head>` de votre fichier `rhtml` :

```
<%= javascript_include_tag "prototype" %>
```

Pour l'ensemble du code utilisé dans ce chapitre, nous avons ajouté l'appel à `javascript_include_tag` dans le fichier global de mise en page *`application.rhtml`*. De cette façon les méthodes AJAX sont accessibles à tous les exemples.

1. http://prototype.conio.net

Vous devez aussi placer le fichier *prototype.js* dans le répertoire *public/javascripts* de votre application. Il est normalement placé là automatiquement si vous utilisez la commande `rails` pour mettre en place l'arborescence de votre application.

link_to_remote

La syntaxe permettant de déclencher un appel AJAX depuis un format `rhtml` peut prendre une forme aussi simple que :

Fichier 18.1

```
<%= link_to_remote("Déclencher Ajax",
                   :update => 'mydiv',
                   :url => { :action => :say_hello }) %>

<div id="mydiv">Ce texte va être modifié</div>
```

La forme la plus simple de l'appel à la méthode `link_to_remote()` utilise trois paramètres.

- le texte utilisé par le lien ;
- l'attribut `id=""` de l'élément de la page HMTL à modifier ;
- l'URL de l'action à solliciter au format `url_for()`.

Quand l'utilisateur clique sur le lien, l'action (`say_hello` dans notre exemple) est déclenchée sur le serveur. La sortie générée par cette action est utilisée pour remplacer le contenu de l'élément `mydiv` de la page courante.

La vue qui gère la réponse ne doit utiliser aucun fichier de mise en page puisque nous ne faisons que mettre à jour un fragment d'une page HTML. L'utilisation des mises en page peut être désactivée en appelant `render()` avec l'option `:layout` à `false` ou en spécifiant d'entrée de jeu que l'action ne doit pas utiliser de mise en page (voir chapitre 17 pour plus d'informations).

Maintenant, définissons une action :

Fichier 18.2

```
def say_hello
  render(:layout => false)
end
```

Et la vue correspondante dans le fichier *say_hello.rhtml :*

Fichier 18.3

```
<em>Tu as le bonjour d'Ajax !</em> (l'ID de la session est <%= session.session_id %>)
```

À l'exécution, le texte « Ce texte va être modifié » qui se trouve dans l'élément `<div>` identifié par l'attribut `id="mydiv"` est modifié (voir figure 18-2) de la façon suivante :

```
Tu as le bonjour d'Ajax ! (l'ID de la Session est <some string>)
```

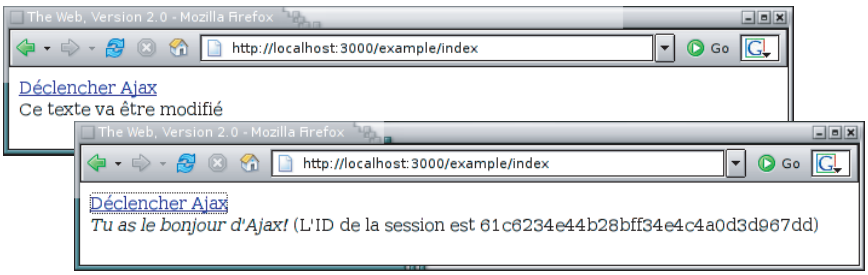


Figure 18-2
Avant et après l'appel de l'action via AJAX

Facile, non ? Notez que nous avons décidé d'afficher l'identifiant de la session pour mettre en avant le fait que la requête `XMLHttpRequest` gère les cookies comme le ferait une requête HTTP classique. Nous sommes donc assurés qu'une application Rails récupérera le bon identifiant de session que l'action soit déclenchée par une requête AJAX ou non.

Derrière le rideau

Examinons maintenant ce qui se cache derrière l'appel à `link_to_remote` de notre exemple.

Tout d'abord, arrêtons-nous un moment sur le code HTML généré par `link_to_remote()`.

```
<a href="#" onclick="new Ajax.Updater('mydiv',
       '/example/say_hello', {asynchronous:true}); return false;">
  Déclencher Ajax
</a>
```

`link_to_remote()` génère une balise HTML `<a>` qui réagit au clic de souris en créant une nouvelle instance de `Ajax.Updater` (lui-même défini dans la bibliothèque Prototype).Cette instance appelle `XMLHttpRequest` en interne, qui à son tour, envoie une requête POST à l'URL indiquée en deuxième paramètre[1]. Le processus est illustré sur la figure 18-3.

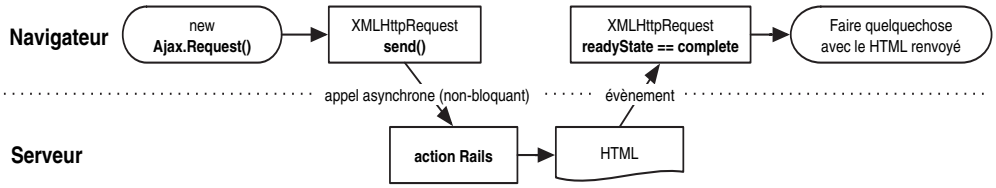


Figure 18-3
XMLHttpRequest connecte navigateurs et serveurs entre eux

1. Pour des raisons de sécurité les URL sollicitées doivent utiliser le même nom de serveur et le même numéro de port que la page qui a recours à `XMLHttpRequest`.

Voyons ce qui se passe côté serveur.

```
127.0.0.1 - - [21/Apr/2005:19:55:26] "POST /example/say_hello HTTP/1.1" 200 51
```

Le serveur Web (ici WEBrick) reçoit une requête HTTP de type POST qui sollicite l'action `/example/say_hello`. Du point de vue du serveur, rien ne différencie cette requête HTTP des autres. Rien de surprenant, étant donné qu'elle *est* exactement comme les autres.

Après avoir exécuté l'action (ici `say_hello()`), la sortie générée est renvoyée à l'objet `XMLHttpRequest` créé en interne par `link_to_remote()`. L'instance `Ajax.Updater` reprend la main côté client et remplace le contenu de l'élément indiqué en premier paramètre (ici `mydiv`) avec les données retournées par l'objet `XMLHttpRequest`. Le navigateur met la page à jour pour qu'elle reflète les changements opérés dans le contenu. L'utilisateur, de son côté, ne verra simplement qu'une seule chose : un changement dans la page.

form_remote_tag()

Vous pouvez aisément modifier un formulaire pour qu'il utilise les techniques AJAX en remplaçant l'appel à `form_tag()` par `form_remote_tag()`.

Cette méthode série automatiquement tous les éléments du formulaire et les envoie au serveur en utilisant `XMLHttpRequest` comme d'habitude. Aucune modification n'est nécessaire en ce qui concerne les actions[1].

Créons maintenant un jeu dont le but est de compléter une phrase simple : l'application dit « Ruby on ... » et l'utilisateur doit deviner le mot manquant. Voici le contrôleur :

Fichier 18.4

```
class GuesswhatController < ApplicationController
  def index
  end
  def guess
    @guess = params[:guess] || ''
    if @guess.strip.match /^rails$/i
      render(:text => "C'est juste !")
    else
      render(:partial => 'form')
    end
  end
end
```

Le format *index.rhtml* ressemble à ceci :

1. Il existe néanmoins une exception : on ne peut pas utiliser un champ de téléchargement de fichier avec `form_remote_tag()` car JavaScript ne peut pas accéder au contenu du fichier. C'est une limitation de JavaScript justifiée par des impératifs de sécurité et de performance.

Fichier 18.5

```
<h3>Devinez quoi !</h3>
<div id="update_div" style="background-color:#eee;">
  <%= render(:partial => 'form') %>
</div>
```

Enfin, la partie principale de ce nouveau jeu révolutionnaire qui va vous rendre riche et célèbre se trouve dans le partiel *_form.rhtml*.

Fichier 18.6

```
<% if @guess %>
  <p>Il semble que '<%=h @guess %>' ne soit pas la bonne réponse</p>
<% end %>
<%= form_remote_tag(:update => "update_div",
                    :url    => { :action => :guess } ) %>
  <label for="guess">Ruby on .....?</label>
  <%= text_field_tag :guess %>
  <%= submit_tag "Envoyer le formulaire avec AJAX" %>
<%= end_form_tag %>
```

Essayez-le. Vous verrez qu'il n'est pas très difficile de trouver la solution comme le montre la figure 18-4.

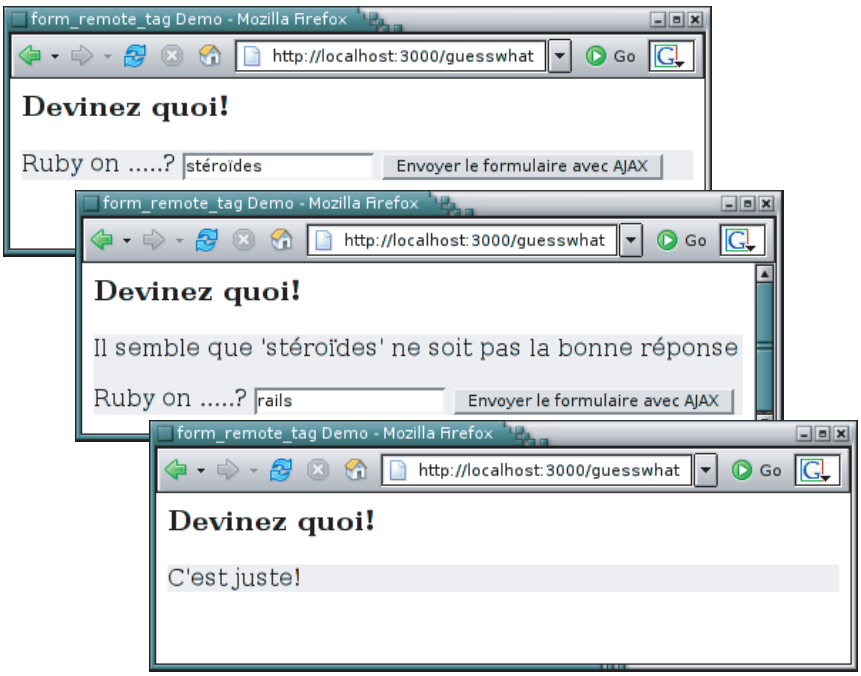


Figure 18-4
Mise à jour d'un formulaire de type AJAX dans une fenêtre existante

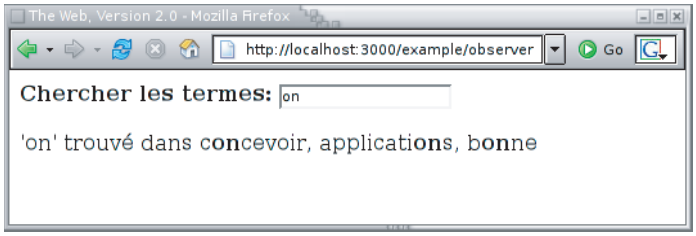
`form_remote_tag()` est une excellente façon d'ajouter des formulaires à la volée dans notre application pour des choses telles qu'un sondage, un chat en ligne, etc. Tout cela sans avoir à changer ni à recharger la page qui les contient.

Les formats partiels vous aident aussi à respecter le principe DRY car vous pouvez les utiliser à la fois dans l'élaboration d'une page complète et comme fragment pour mettre à jour les pages qui ont recours à des actions version AJAX. Aucun changement n'est nécessaire.

Observateurs (Observers)

Un observateur permet de faire appel à des actions AJAX lorsque l'utilisateur remplit un formulaire ou le modifie. C'est très pratique, par exemple, pour construire un mécanisme de recherche par mots-clés qui réagit en temps réel.

Figure 18-5
Construire une recherche par mots-clés avec un observateur



Fichier 18.7

```
<label for="search">Chercher les termes :</label>
<%= text_field_tag :search %>
<%= observe_field(:search,
                  :frequency => 0.5,
                  :update    => :results,
                  :url       => { :action => :search }) %>
<div id="results"></div>
```

L'observateur surveille toute modification intervenant sur un champ particulier du formulaire avec une périodicité de `:frequency` secondes. Par défaut, `observe_field` transmet à l'action la valeur brute du champ surveillé dans les données de la requête POST. Vous pouvez accéder à ces données depuis le contrôleur avec l'appel `request.raw_post`.

Maintenant que l'observateur est en place, implémentons l'action `search`. Nous voulons mettre en œuvre une recherche dans une liste de mots prédéfinis présents dans un tableau, le mot recherché étant présenté en surlignage dans les résultats.

Fichier 18.2

```
WORDLIST = %w(Rails is a full-stack, open-source web framework in Ruby
  for writing real-world applications with joy and less code than most
  frameworks spend doing XML sit-ups)
```

Fichier 18.2

```
def search
  @phrase = request.raw_post || request.query_string
  matcher = Regexp.new(@phrase)
  @results = WORDLIST.find_all { |word| word =~ matcher }
  render(:layout => false)
end
```

La vue, *search.rhtml*, ressemble à ceci :

Fichier 18.8

```
<% if @results.empty? %>
  '<%=h @phrase %>' not found !
<% else %>
  '<%=h @phrase %>' found in
  <%= highlight(@results.join(', '), @phrase) %>
<% end %>
```

Pointez votre navigateur vers l'action `observer` et vous verrez apparaître un champ de saisie qui déclenche une recherche assistée en temps réel (voir figure 18-5). Notez que dans cet exemple la fonction de recherche supporte les expressions régulières.

La ligne `@phrase = request.raw_post || request.query_string` vous permet de saisir directement dans le navigateur une URL telle que `/controller/search?ruby`. Cela peut être très pratique en phase de test où, en l'absence de données brutes en provenance du formulaire, l'action utilise à la place la chaîne de caractères de l'URL pour la recherche.

L'action invoquée par l'observateur ne doit pas être excessivement complexe. Elle peut en effet être sollicitée fréquemment suivant la vitesse de frappe de l'utilisateur et la périodicité de la surveillance. Autrement dit, évitez les requêtes lourdes vers la base de données ou toute opération coûteuse en temps. Vos utilisateurs vous en sauront gré aussi car vous préserverez du même coup la réactivité de l'interface utilisateur.

Mises à jour périodique

Le troisième assistant AJAX, `periodically_call_remote()`, vous sera d'un grand secours si vous souhaitez mettre périodiquement à jour certaines parties de vos pages.À titre d'exemple nous allons bâtir une application qui énumère les processus en cours d'exécution sur le serveur et rafraîchit la liste des processus après quelques secondes. Cet exemple utilise la commande `ps`, ce qui le cantonne aux serveurs de type Unix. En mettant la commande entre guillemets inversés, le résultat est retourné sous la forme d'une chaîne de caractères. Voici le contrôleur de l'application :

Fichier 18.2

```
def periodic
  # No action...
end
# Renvoie la liste des processus (code spécifique à Unix)
def ps
  render(:text => "<pre>" + CGI::escapeHTML(`ps -a`) + "</pre>")
end
```

Et voici le format *periodic.rhtml* qui contient l'appel à `periodically_call_remote()` :

Fichier 18.9

```
<h3>Processus du serveur:</h3>
<div id="process-list" style="background-color:#eee;">
</div>
<%= periodically_call_remote(:update    => 'process-list',
                             :url       => { :action => :ps },
                             :frequency => 2 )%>
```

Si vous avez payé le supplément pour obtenir la version avec « serveur Web embarqué » de cet ouvrage, vous verrez la figure 18-6 se mettre à jour toutes les deux secondes (la valeur dans la colonne TIME du processus « ruby script/server » devrait croître à chaque itération). Si vous avez juste la version papier ou PDF du livre vous devez nous croire sur parole.

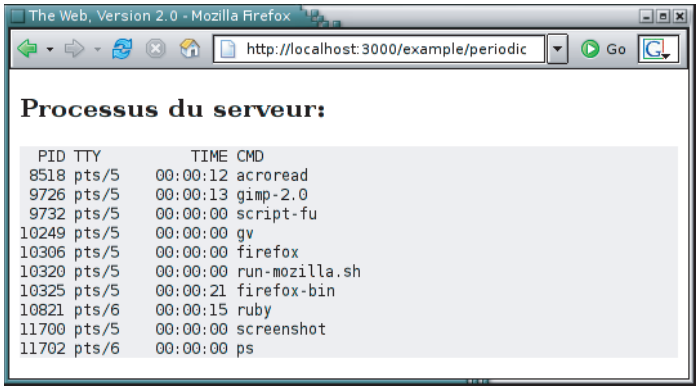


Figure 18-6
Rester à jour avec `periodically_call_remote`

L'interface utilisateur, revisitée

Les applications Web offrent traditionnellement une interface utilisateur moins interactive que celle des applications de bureau. Jusqu'à très récemment il était difficile d'y remédier. Avec l'émergence du Web version 2.0, cela peut et doit changer, puisque, grâce à AJAX, nous avons maintenant une batterie de mécanismes permettant de contrôler finement nos pages Web.

La bibliothèque Prototype aide notre application à communiquer de façon intuitive avec l'utilisateur. Et en plus, c'est vraiment agréable !

En plus des appels AJAX, la bibliothèque Prototype offre une panoplie d'objets très utiles qui rendent la vie plus facile aux développeurs et qui améliorent encore la relation entre votre application et ses utilisateurs.

Les fonctionnalités offertes par la bibliothèque Prototype se divisent en trois groupes :

- les appels AJAX (que nous venons juste de traiter) ;
- les manipulations du Document Object Model (DOM) ;
- les effets visuels.

Manipulation du Document Object Model (DOM)

Les manipulations du modèle objet du document (DOM) en JavaScript sont pénibles et mal conçues. C'est pour cette raison que Prototype fournit des raccourcis Ruby pour un certain nombre d'opérations souvent utilisées. Ces fonctions sont toutes écrites en JavaScript et sont destinées à être invoquées depuis les pages du navigateur.

`$()`

Passez en paramètre à `$()` une chaîne de caractères. Elle renverra l'élément DOM dont l'identifiant est spécifié. Celui-ci est retourné si aucun élément correspondant n'est trouvé. Ce comportement permet de transmettre en paramètre l'`id=""` d'un élément ou l'élément lui-même, et d'obtenir un élément en retour.

```
        $('mydiv').style.border = "1px solid red"; /* définit le liseré de mydiv
*/
```

`Element.toggle()`

`Element.toggle()` montre ou cache le ou les éléments indiqués. En interne, la valeur de l'attribut CSS display bascule entre les valeurs `'inline'` et `'none'`.

```
        Element.toggle('mydiv');                /* bascule mydiv */
        Element.toggle('div1', 'div2', 'div3'); /* bascule div1-div3 */
```

`Element.show()`

`Element.show()` s'assure que tous les éléments passés en paramètres sont visibles.

```
        Element.show('warning');      /* montre les éléments dont l'id est
'warning' */
```

`Element.hide()`

L'inverse de `Element.show()`.

`Element.remove()`

`Element.remove()` détruit totalement un élément du DOM.

```
Element.remove('mydiv');      /* efface complètement mydiv */
```

Méthodes d'insertion :

Les diverses méthodes d'insertion permettent d'ajouter facilement des fragments HTML à des éléments déjà existants. Elles sont abordées à la section *Techniques de remplacement*, page 427.

Effets visuels

Comme AJAX travaille en tâche de fond, l'utilisateur ne le perçoit pas. Le serveur peut recevoir une requête AJAX sans que l'utilisateur le sache. Le navigateur n'indique pas non plus qu'une requête AJAX est en cours. L'utilisateur pourrait ainsi cliquer sur un bouton pour détruire une entrée dans une liste de tâches. L'action serait déclenchée en arrière-plan mais, sans retour d'information, comment l'utilisateur pourrait-il savoir ce qui s'est passé ? Et, en général, quand il ne se passe rien, un utilisateur à tendance à cliquer sur le même bouton encore et encore.

Notre travail en tant que développeur est de renseigner l'utilisateur sur l'état d'avancement de la requête en cours quand le navigateur ne le fait pas de lui-même. Une information visuelle doit lui indiquer que quelque chose est en train de se passer. La première méthode consiste à utiliser diverses techniques de manipulation DOM pour modifier la page mais cela peut s'avérer insuffisant.

Prenons l'exemple d'un appel à `link_to_remote()` qui détruit un enregistrement de la base de données et l'élément DOM correspondant, entraînant la disparition instantanée de l'élément sur la page HTML. Dans une application de bureau traditionnelle ce comportement est attendu, mais dans le cas d'une application Web l'utilisateur peut ne pas voir ce changement parce qu'il ne s'y attend pas.

C'est pourquoi il est fortement conseillé dans un deuxième temps, d'avoir recours à des effets visuels montrant que le changement est en train de s'effectuer, par exemple, en disparaissant progressivement ou en se dissolvant dans la page.

Les effets visuels de Rails sont regroupés dans une bibliothèque JavaScript, *`effects.js`*. Étant donné que cette bibliothèque dépend de *`prototype.js`*, vous devez inclure les deux dans vos pages HTML si vous voulez utiliser les effets visuels sur votre site (probablement en utilisant le fichier de mise en page global).

```
<%= javascript_include_tag "prototype", "effects" %>
```

Il existe deux types d'effets : les effets ponctuels et les effets répétables.

Effets ponctuels

Ces effets sont utilisés pour présenter un message à l'utilisateur : *quelque chose vient de disparaître, de changer ou d'être créé*. Tous ces effets prennent un seul paramètre à savoir un élément de votre page. Vous devez utiliser l'identifiant qui caractérise l'élément pour lui appliquer l'effet comme dans : `new Effect.Fade('id_of_an_element')`. Si vous utilisez un effet dans le cadre du traitement des événements de l'élément vous pouvez aussi utiliser la forme `new Effect.Fade(this)`, ce qui évite d'avoir à nommer l'élément explicitement.

`Effect.Appear()`

Cet effet change progressivement l'opacité de l'élément spécifié de 0 % à 100 %, le faisant ainsi apparaître progressivement.

`Effect.Fade()`

Cet effet est l'inverse de `Effect.Appear()` ; l'élément disparaît progressivement de l'écran et la propriété CSS display passe à la valeur *none* à la fin afin de retirer l'élément du flux normal de la page.

`Effect.Highlight()`

Utilise la fameuse technique *Yellow Fade Technique*[1] sur l'élément spécifié, passant ainsi progressivement l'arrière-plan du jaune au blanc. C'est une excellente façon de mettre en évidence quelque chose qui vient d'être mis à jour non seulement sur la page du navigateur mais aussi sur le serveur.

`Effect.Puff()`

Fait en sorte qu'un élément disparaisse dans un joli petit nuage de fumée. L'élément disparaît progressivement en réduisant sa taille. À la fin de l'animation, la propriété CSS display est positionnée à *none* (voir figure 18-7).

`Effect.Squish()`

Fait disparaître l'élément en le réduisant progressivement.

Les copies d'écran de la figure 18-7 ont été générées avec les formats qui suivent. Le code en tête du format définit un style alternatif pour les carrés de la grille. La boucle qui se trouve en bas crée 16 carrés. Quand un lien *Détruire* est activé, l'action `destroy` du contrôleur est appelée. Dans cet exemple, l'action ne fait rien mais, dans une application réelle, elle détruirait probablement quelque chose dans la base de données. Quand l'action s'achève, l'effet `Puff` est invoqué sur le carré cliqué, le faisant ainsi disparaître.

1. Popularisé par le site de 37signals ; voir http://www.37signals.com/svn/archives/000558.php.

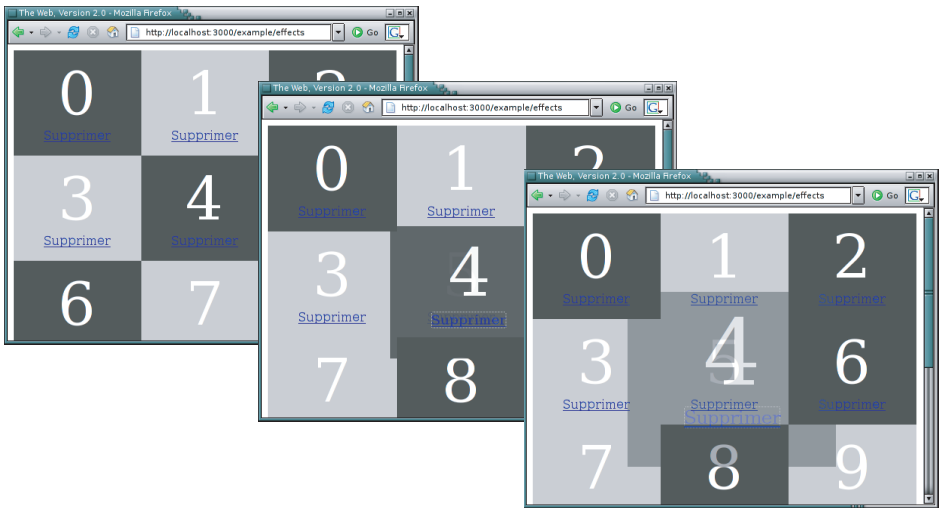


Figure 18-7
Les carrés viennent et s'en vont ...

Fichier 18.10

```
<% def style_for_square(index)
     color = (index % 2).zero? ? "#444" : "#ccc"
     %{ width: 150px;  height: 120px; float: left;
        padding: 10px; color: #fff; text-align:center;
        background: #{color} }
   end
%>
<% 16.times do |i| %>
  <div id="mydiv<%= i %>" style="<%= style_for_square(i) %>">
    <div style="font-size: 5em;"><%= i %></div>
    <%= link_to_remote("Supprimer",
                       :complete => "new Effect.Puff('mydiv#{i}')",
                       :url => { :action => :destroy, :id => i }) %>
  </div>
<% end %>
```

Effets répétables

`Effect.Scale()`

Cet effet change la taille d'un élément. Si vous changez la taille d'un `<div>`, tous les éléments qui s'y trouvent doivent utiliser l'unité *em* pour la hauteur et la largeur. À noter qu'il n'est pas obligatoire de donner explicitement hauteur et largeur.

Changeons la taille d'une image.

```
<%= image_tag("image1",
              :onclick  => "new Effect.Scale(this, 100)") %>
```

Vous pouvez aussi le faire sur un texte si vous utilisez l'unité *em* pour la taille de la fonte.

```
<%= content_tag("div",
                "Texte dont la taille va changer.",
                :style    => "font-size:1.0em; width:100px;",
                :onclick  => "new Effect.Scale(this, 100)") %>
```

`Element.setContentZoom()`

Cet effet modifie sans animation la taille d'un texte ou d'autres éléments qui utilisent l'unité *em*.

```
<div id="outerdiv"
  style="width:200px; height:200px; border:1px solid red;">
  <div style="width:10em; height:2em; border:1px solid blue;">
    Premier div imbriqué
  </div>
  <div style="width:150px; height: 20px; border:1px solid blue;">
    Second div imbriqué
  </div>
</div>
<%= link_to_function("Small",  "Element.setContentZoom('outerdiv', 75)")
%>
<%= link_to_function("Medium", "Element.setContentZoom('outerdiv',
100)") %>
<%= link_to_function("Large",  "Element.setContentZoom('outerdiv', 125)")
%>
```

Remarquez que la taille du deuxième élément `<div>` ne change pas car il n'utilise pas l'unité *em*.

Techniques avancées

Passons en revue quelques techniques AJAX plus sophistiquées.

Techniques de remplacement

Comme indiqué plus haut, la bibliothèque Prototype fournit plusieurs techniques de remplacement avancées qui font plus que substituer un élément par un autre. On les utilise via les méthodes de la classe `Insertion`.

`Insertion.Top()`

Insère un fragment HTML après le tout début d'un élément.

```
            new Insertion.Top('mylist', '<li>Hé, je suis le premier de la liste !</
li>');
```

`Insertion.Bottom()`

Insère un fragment HTML immédiatement avant la fin d'un élément. Vous pouvez par exemple l'utiliser pour insérer une nouvelle ligne à la fin d'un élément `<table>` ou un nouvel élément à la fin d'une liste de type `<ol>` ou `<ul>`.

```
             new Insertion.Bottom('mytable', '<tr><td>Voici une nouvelle ligne !</
td></tr>');
```

`Insertion.Before()`

Insère un fragment HTML avant le début d'un élément.

```
             new Insertion.Before('mypara', '<h1>Je suis dynamique !</h1>');
```

`Insertion.After()`

Insère un fragment HTML après la fin d'un élément.

```
            new Insertion.After('mypara', '<p>Et un nouveau paragraphe, un !</p>');
```

Quelques précisions sur les procédures de rappel

Vous pouvez utiliser quatre procédures de rappel (ou callbacks) JavaScript avec les méthodes `link_to_remote()`, `form_remote_tag()` et `observe_`*`xxx`*. Ces procédures de rappel ont automatiquement accès à une variable JavaScript appelée `request` qui contient l'objet `XMLHttpRequest` correspondant.

`:loading()`

Invoquée quand l'objet `XMLHttpRequest` commence à envoyer des données vers le serveur (c'est-à-dire, quand il fait un appel).

`:loaded()`

Invoquée lorsque toutes les données ont été envoyées au serveur et que l'objet `XMLHttpRequest` attend la réponse du serveur.

`:interactive()`

Cet événement est déclenché quand les données commencent à arriver du serveur. Il est à noter que l'implémentation de cet événement varie en fonction du navigateur.

`:complete()`

Invoquée quand toutes les données en provenance du serveur ont été reçues et que l'appel est terminé.

En l'état actuel des choses, il est probablement sage d'éviter les procédures de rappel `:loaded()` et `:interactive()` car elles se comportent de façon très différente d'un navigateur à l'autre. `:loading()` et `:complete()` fonctionnent avec tous les navigateurs supportés et seront toujours appelées une et une seule fois.

`link_to_remote()` propose plusieurs paramètres supplémentaires pour une plus grande flexibilité.

`:confirm`

Utilise une boîte de dialogue pour confirmation comme `:confirm` dans `link_to()`.

`:condition`

Fournit une expression JavaScript qui est évaluée lorsqu'on clique sur le lien ; la requête ne sera exécutée que si l'expression retourne la valeur true.

`:before`, `:after`

Évalue une expression JavaScript immédiatement avant et/ou après l'appel AJAX (notez que `:after` n'attend pas le retour de l'appel ; pour cela utilisez la procédure de rappel `:complete`).

L'objet `request` propose aussi quelques méthodes utiles.

`request.responseText`

Retourne le corps de la réponse fournie par le serveur (sous la forme d'une chaîne de caractères).

`request.status`

Retourne le code du statut HTTP donné par le serveur (par exemple, 200 signifie un succès, 404 une page introuvable)[1].

1. Voir le chapitre 10 de la RFC 2616 pour les valeurs que peut prendre le code de retour d'une requête HTTP. Il est disponible en ligne sur http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

`request.getResponseHeader()`

Retourne la valeur de l'en-tête HTTP (header) indiqué dans la réponse renvoyée par le serveur.

Indicateurs de progression

Vous pouvez utiliser les procédures de rappel pour donner aux utilisateurs le sentiment que quelque chose est en train de se produire.

Regardez plutôt cet exemple.

```
<%= text_field_tag :search %>
<%= image_tag("indicator.gif",
              :id    => 'search-indicator',
              :style => 'display:none') %>
<%= observe_field("search",
      :update   => :results,
      :url      => { :action => :search},
      :loading  => "Element.show('search-indicator')",
      :complete => "Element.hide('search-indicator')") %>
```

L'image *`indicator.gif`* est affichée uniquement pendant que l'appel est actif. Pour une meilleure esthétique, utilisez une image animée[1].

Pour la complétion automatique d'un champ texte avec `text_field()`, l'indicateur de progression est déjà fourni.

```
<%= text_field(:items,
               :description,
               :remote_autocomplete => { :action => :autocomplete },
               :indicator => "/chemin/vers/image") %>
```

Mises à jour multiples

Si vous vous reposez énormément sur le serveur pour procéder à des mises à jour des pages côté client et que vous avez besoin de plus de flexibilité que ce que propose l'identification des éléments par leur `:update => 'elementid'`, les procédures de rappel peuvent peut-être vous aider.

L'astuce consiste à faire envoyer du JavaScript par le serveur dans la réponse AJAX. Comme ce code JavaScript a un accès complet au modèle DOM de la page, il peut mettre à jour autant de parties de la page HTML que nécessaire. Pour profiter de cette possibilité, utilisez

1. Vous pouvez vous inspirer des images utilisées par les autres sites Web pour indiquer que le chargement d'une page est en cours.

`:complete => "eval(request.responseText)"` au lieu de `:update`. Vous pouvez générer du JavaScript dans vos vues pour qu'il soit ensuite délivré et exécuté au niveau du navigateur.

À titre d'exemple, utilisons des effets visuels au hasard sur plusieurs éléments d'une page. Nous avons d'abord besoin d'un contrôleur.

Fichier 18.2

```
def multiple
end
def update_many
  render(:layout => false)
end
```

Rien de fracassant à ce niveau. Mais le fichier *multiple.rhtml* est plus intéressant.

Fichier 18.11

```
<%= link_to_remote("Mises à jour multiples",
                   :complete => "eval(request.responseText)",
                   :url => { :action => :update_many }) %>

<hr/>
<% style = "float:left; width:100px; height:50px;" %>

<% 40.times do |i|
     background = "text-align: center; background-color:##{("%02x" % (i*5))*3};"
%>

  <%= content_tag("div",
                  "Je suis div #{i}",
                  :id => "div#{i}",
                  :style => style + background) %>
<% end %>
```

Il crée 40 éléments `<div>` et le code `eval(request.responseText)` sur la deuxième ligne nous permet de générer du JavaScript dans le format *update_many.rhtml* comme suit :

Fichier 18.12

```
<% 3.times do %>
  new Effect.Fade('div<%= rand(40) %>');
<% end %>
```

À chaque fois que vous cliquez sur « Mises à jour multiples » le serveur renvoie trois lignes de JavaScript qui, une fois exécutées, vont faire progressivement disparaître trois éléments `<div>` pris au hasard !

Pour insérer sans problème un fragment HTML quel qu'il soit, utilisez l'assistant `escape_javascript()`. De cette façon vous êtes sûr que tous les caractères `'` et `"` ainsi que les

retours à la ligne seront exclus correctement en vue de construire une chaîne de caractères JavaScript valide.

```
        new Insertion.Bottom('mytable',
          '<%= escape_javascript(render(:partial => "ligne")) %>');
```

Si une vue renvoie du JavaScript à exécuter par le navigateur, vous devez vous préoccuper de ce qui se passera si une erreur survient lors du rendu de la page. Par défaut, Rails renvoie une page avec une erreur HTML, ce qui n'est pas vraiment ce qu'il nous faut dans ce cas puisqu'il s'agit d'une erreur JavaScript.

Au moment où nous écrivons ces lignes, l'équipe Rails travaille à l'ajout d'un gestionnaire d'erreur pour `link_to_remote()` et `form_remote_tag()`. Consultez le document RDoc de Rails pour les derniers détails.

Mise à jour automatique d'une liste

Un des usages les plus répandus d'AJAX consiste à mettre une liste à jour sur le navigateur. Au fur et à mesure que l'utilisateur ajoute ou enlève des éléments, la liste se met à jour sans rafraîchir la page en entier. À titre d'exercice, écrivons le code qui réalise cette opération. Cette fonctionnalité est non seulement utile mais elle nous permet aussi d'appliquer plusieurs techniques apprises dans ce chapitre.

Notre application est (une fois de plus) un gestionnaire de tâches. Elle affiche une liste de choses à faire et un formulaire qui permet d'ajouter de nouvelles tâches. Commençons par écrire la version qui utilise des formulaires HTML conventionnels sans AJAX.

Plutôt que de se soucier des tables de la base de données, nous allons utiliser un modèle dont la classe travaille uniquement en mémoire. Voici *item.rb* tel qu'il est stocké dans *app/models*.

Fichier 18.13

```
class Item
  attr_reader :body
  attr_reader :posted_on
  FAKE_DATABASE = []

  def initialize(body)
    @body = body
    @posted_on = Time.now
    FAKE_DATABASE.unshift(self)
  end
  def self.find_recent
     FAKE_DATABASE
  end

  # Populate initial items
```

```
  new("Nourrir le chat")
  new("Laver la voiture")
  new("Vendre ma start-up à Google")
end
```

Le contrôleur propose deux actions, l'une énumère les tâches courantes, l'autre ajoute une tâche à la liste.

Fichier 18.14

```
class ListNoAjaxController < ApplicationController
  def index
    @items = Item.find_recent
  end
  def add_item
    Item.new(params[:item_body])
    redirect_to(:action => :index)
  end
end
```

La vue montre une simple liste et un formulaire de saisie.

Fichier 18.15

```
<ul id="items">
<%= render(:partial => 'item', :collection => @items) %>
</ul>
<%= form_tag(:action => "add_item") %>
   <%= text_field_tag('item_body') %>
   <%= submit_tag("Ajouter une tâche") %>
<%= end_form_tag %>
```

Elle utilise un partiel très simple pour chaque ligne.

Fichier 18.16

```
<li>
  <p>
    <%= item.posted_on.strftime("%H:%M:%S") %>:
    <%= h(item.body) %>
  </p>
</li>
```

Maintenant ajoutons le support AJAX à cette application. Nous allons modifier le formulaire pour soumettre la nouvelle action via `XMLHttpRequest`, action qui placera la nouvelle tâche au sommet de la liste déjà visible.

```
  <ul id="items">
    <%= render(:partial => 'item', :collection => @items) %>
  </ul>
<%= form_remote_tag(:url => { :action => "add_item" },
                    :update => "items",
                    :position => :top) %>
  <%= text_field_tag('item_body') %>
  <%= submit_tag("Add Item") %>
  <%= end_form_tag %>
```

Nous modifions ensuite le contrôleur pour qu'il effectue le rendu de la nouvelle tâche dans la méthode `add_item`. Remarquez au passage la façon dont la vue et l'action partagent le même partiel. C'est une façon de faire assez répandue ; la vue utilise un partiel pour effectuer le rendu de la liste initiale et le contrôleur l'utilise aussi pour afficher une nouvelle tâche quand elle est ajoutée.

Fichier 18.17

```
def add_item
  item = Item.new(params[:item_body])
  render(:partial => "item", :object => item, :layout => false)
end
```

Mais nous pouvons faire encore mieux. Mettons quelque chose de plus abouti dans les mains de l'utilisateur. Nous allons utiliser les procédures de rappel `:loading` et `:complete` pour donner un retour visuel à nos utilisateurs sur la prise en compte de leur demande.

- Quand l'utilisateur clique sur le bouton **`Ajouter une tâche`**, celui-ci est désactivé et un message apparaît indiquant que la demande est en cours de traitement.
- Quand le serveur retourne la réponse, nous utilisons l'effet Yellow Fade pour montrer la tâche nouvellement créée. Pour finir, le message est supprimé, le bouton est réactivé, le champ texte est effacé et le focus repositionné sur le champ texte prêt pour la saisie d'une nouvelle tâche.

Ces améliorations vont nécessiter deux fonctions JavaScript. Nous les placerons dans la section `<script>` de l'en-tête de page HTML. Plutôt que de définir cet en-tête dans chaque format de page nous allons utiliser la déclaration `content_for` pour stocker du texte dans une variable d'instance. On peut ensuite insérer dans le format le contenu de cette variable dans l'en-tête de page HTML. De cette façon, le format de chaque action peut personnaliser le format commun.

Dans le format *`index.rhtml`* nous allons utiliser la méthode `content_for()` pour stocker dans la variable `@contents_for_page_scripts` le texte correspondant au code de deux fonctions JavaScript. Quand Rails procède au rendu de ce format, les deux fonctions JavaScript seront

automatiquement incluses. Nous avons aussi ajouté des procédures de rappel dans l'appel à `form_remote_tag` et créé le message à afficher lorsque le traitement du formulaire est en cours.

Fichier 18.18

```
<% content_for("page_scripts") do -%>
function item_added() {
  var item = $('items').firstChild;
  new Effect.Highlight(item);
  Element.hide('busy');
  $('form-submit-button').disabled = false;
  $('item-body-field').value = '';
  Field.focus('item-body-field');
}
function item_loading() {
  $('form-submit-button').disabled = true;
  Element.show('busy');
}
<% end -%>
<ul id="items">
<%= render(:partial => 'item', :collection => @items) %>
</ul>
<%= form_remote_tag(:url => { :action => "add_item" },
                    :update   => "items",
                    :position => :top,
                    :loading  => 'item_loading()',
                    :complete => 'item_added()') %>
   <%= text_field_tag('item_body', '', :id => 'item-body-field') %>
   <%= submit_tag("Ajouter une tâche", :id => 'form-submit-button') %>
    <span id='busy' style="display: none">En cours...</span>
<%= end_form_tag %>
```

Ensuite, le contenu de la variable d'instance `@contents_of_page_scripts` est inséré dans l'entête de la page HTML.

Fichier 18.19

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <%= javascript_include_tag("prototype", "effects") %>
  <script type="text/javascript"><%= @content_for_page_scripts %></script>
  <title>Ma liste de tâches</title>
</head>
<body>
<%= @content_for_layout %>
</body>
```

En général, il est recommandé de commencer le développement d'une application en utilisant des pages HTML conventionnelles puis d'ajouter le support AJAX dans un second temps. Cela permet de se concentrer sur la logique de l'application dans un premier temps puis de travailler sur la présentation par la suite.

Utiliser des effets sans AJAX

Produire des effets dans les pages HTML sans avoir recours à AJAX est assez délicat. Bien qu'il soit tentant d'utiliser l'événement `window.onload` pour se passer d'AJAX, vos effets n'apparaîtront à l'écran qu'après le chargement de tous les autres éléments de la page y compris les images.

Placer une balise `<script>` directement après l'élément HTML concerné est aussi une alternative, mais sur certains navigateurs et suivant le contenu de la page, des problèmes de présentation peuvent survenir. Si tel est le cas, vous pouvez essayer d'insérer la balise `<script>` à la fin de la page HTML.

L'extrait RHTML suivant applique l'effet Yellow Fade à un élément particulier sans avoir recours à AJAX.

Fichier 18.20

```
<div id="mydiv">Du contenu</div>
<script type="text/javascript">
  new Effect.Highlight('mydiv');
</script>
```

Test

Le test de vos fonctions et formulaires utilisant AJAX n'est pas différent de ce qui se fait pour des liens et des formulaires HTML conventionnels. Il existe une méthode particulière pour simuler des appels aux actions exactement comme s'ils avaient été générés par la bibliothèque Prototype. La méthode `xml_http_request()` (ou `xhr()` en raccourci) encapsule les méthodes `get()`, `post()`, `put()`, `delete()` et `head()`, permettant ainsi au code de test d'invoquer des actions du contrôleur comme s'il s'agissait d'une requête en provenance du code JavaScript du navigateur. À titre d'exemple, un test peut utiliser le code suivant pour invoquer l'action `index` d'un contrôleur appelé post :

```
xhr :post, :index
```

L'appel à xhr a pour effet de positionner le résultat de `request.xhr?` à `true` (voir section *Appelée depuis AJAX ?*, page 436).

Si vous envisagez d'ajouter des tests au niveau de votre navigateur ou des tests fonctionnels, nous vous conseillons de regarder l'outil Selenium[1]. Cet outil permet de vérifier par exemple

1. http://selenium.thoughtworks.com/

les changements DOM directement dans votre navigateur. Pour des tests unitaires JavaScript, JsUnit est un bon choix[1].

Si vous rencontrez des comportements inattendus dans votre application, n'oubliez pas d'examiner la console JavaScript de votre navigateur. Malheureusement, tous les navigateurs n'en disposent pas. À noter que les utilisateurs de Firefox et Mozilla peuvent avoir recours à Venkman[2], excellent debugger JavaScript.

Compatibilité ascendante

Rails propose plusieurs fonctionnalités permettant d'exécuter des applications à base d'AJAX sur des navigateurs qui ne disposent pas du support JavaScript.

Il vous faut cependant décider assez tôt dans le processus de développement si vous souhaitez y avoir recours car cela a des implications profondes sur le développement du code.

Appelée depuis AJAX ?

Pour déterminer si une action est appelée depuis la bibliothèque Prototype, utilisez la méthode `request.xml_http_request?` ou son raccourci `request.xhr?`.

Fichier 18.2

```
def checkxhr
  if request.xhr?
    render(:text => "21st century Ajax style.", :layout => false)
  else
    render(:text => "Ye olde Web.")
  end
end
```

Voici le format *check.rhtml* correspondant :

Fichier 18.21

```
<%= link_to_remote('Ajax..',
                   :complete => 'alert(request.responseText)',
                   :url => { :action => :checkxhr }) %>

<%= link_to('Pas Ajax...', :action => :checkxhr) %>
```

Ajouter des liens HTML standards à AJAX

Pour utiliser des liens HTML standards dans vos appels à `link_to_remote`, il suffit d'ajouter un paramètre `:href => URL`. Les navigateurs qui ont désactivé JavaScript utiliseront simplement le lien HTML standard au lieu de l'appel AJAX. Il est très important de procéder de cette

1. http://www.edwardh.com/jsunit/
2. http://www.mozilla.org/projects/venkman/

façon si vous voulez que votre site reste accessible au plus grand nombre et notamment par les personnes mal voyantes qui utilisent des navigateurs très spécifiques.

Fichier 18.22

```
<%= link_to_remote("Fonctionne aussi sans JavaScript ...",
                   { :update => 'mydiv',
                     :url    => { :action => :say_hello } },
                   { :href   => url_for( :action => :say_hello ) } ) %>
```

Pour les appels à `form_remote_tag()` il n'y a aucune disposition particulière à prendre étant donné que le formulaire HTML comprend déjà un attribut classique `action=` qui déclenche l'action spécifiée dans le paramètre `:url`. Si JavaScript est activé, l'appel AJAX est utilisé, sinon, une requête HTTP POST classique est émise par le navigateur. Si vous souhaitez déclencher des actions différentes suivant que JavaScript est supporté ou non, ajoutez un paramètre `:html => { :action => URL, : method => 'post' }`. Par exemple, dans l'extrait de code qui suit, le formulaire invoque l'action `guess` si JavaScript est activé et l'action `post_guess` dans le cas contraire.

Fichier 18.23

```
<%= form_remote_tag(
       :update => "update_div",
       :url    => { :action => :guess },
       :html   => {
         :action => url_for( :action => :post_guess ),
         :method => 'post' } ) %>
  <% # ...  %>
<%= end_form_tag %>
```

Évidemment, cela ne vous dispense pas d'accorder une certaine attention à la façon dont vous effectuez le rendu des pages. Vos actions doivent savoir de quelle façon elles sont appelées et agir en conséquence.

Le blues du bouton arrière

Comme tout le monde le sait, le bouton arrière de votre navigateur permet de revenir à l'intégralité de la page précédemment affichée (ce qui n'est en général possible que si un lien HTML classique a été utilisé).

Vous devez prendre en compte cet aspect quand vous spécifiez l'enchaînement des pages dans votre application, notamment dans la façon de grouper les objets des pages. Un parent et ses objets enfants appartiennent typiquement au même groupe logique, alors que les parents se trouvent normalement dans des groupes distincts. Il est en général conseillé d'utiliser des liens non-AJAX pour naviguer entre les groupes et des fonctions AJAX uniquement au sein d'un groupe. Par exemple, dans le cas d'une application de blog, il vaut mieux utiliser un lien HTML classique pour passer de la page d'accueil d'un blog à un article particulier (pour que

le bouton arrière revienne à la page d'accueil) et utiliser AJAX pour poster un commentaire lié à un article[1].

Web V2.1

AJAX évolue rapidement et Rails est à la pointe dans ce domaine. Il est donc difficile de produire une documentation parfaitement à jour dans un livre tel que celui-ci.

Gardez un œil ouvert sur les évolutions de Rails et du support AJAX. Ne serait-ce que pendant les quelques mois nécessaires à l'écriture de ce livre, nous avons vu émerger des techniques AJAX comme la complétion des champs texte (à la manière de Google Suggest), l'indicateur de progression lors des chargements de fichier, le support du glisser-déposer (drag-and-drop), les listes qui peuvent être réordonnées à l'écran, etc.

La visite régulière du site de Thomas Fuch http://script.aculo.us/ est une bonne façon de se tenir informé des dernières évolutions et d'expérimenter tous les effets AJAX.

1. C'est ce que fait le fameux blog Typo qui est écrit en Rails. Visitez le site http://typo.leetsoft.com/ pour plus d'informations.