6

Les « frameworks » Ajax

Au cours des chapitres précédents, nous avons eu constamment besoin de définir, combiner ou réutiliser des composants JavaScript, certains pour définir des comportements (suggestion de saisie, récupération de flux RSS, etc.) ou des composants graphiques (boîte d'information, liste pop-up, etc.), et d'autres pour encapsuler les différences d'implémentation des navigateurs, fort nombreuses en ce qui concerne le HTML dynamique, ou encore étendre par des fonctions utiles le DOM et les objets JavaScript natifs.

De nombreuses bibliothèques de tels composants fleurissent sur le marché, sans compter les bibliothèques de composants DHTML existant déjà depuis de nombreuses années. Nous allons les étudier dans ce chapitre, en nous concentrant sur les plus connues : prototype, qui fait partie du framework Ruby on Rails, dojo, une bibliothèque extrêmement vaste, et Yahoo User Interface, récemment publiée.

Boîtes à outils, bibliothèques et frameworks

Le mot framework est aujourd'hui à la mode, les médias ne cessant de l'employer. Certains produits se vantent d'être des « frameworks Ajax », tandis que d'autres, plus modestes quoique parfois bien plus puissants, se qualifient plus justement de bibliothèques ou de boîtes à outils. Il serait peut-être utile de commencer par distinguer ces termes.

L'expression *boîte à outils* (toolkit) a deux sens :

- Elle a d'abord désigné un ensemble de composants mis à la disposition du développeur : classes (ou objets) utilisables dans le code, exécutables, comme `lex` et `yacc` sous Unix ou le programme Perl `jsdoc.pl`, vu au chapitre 3, qui génère la documentation JavaScript à partir des commentaires jsdoc. Les éléments qui se révèlent

utiles et fréquemment employés sont ajoutés au fur et à mesure à la boîte à outils, pour être en quelque sorte tenus sous la main, si bien que l'ensemble peut être un peu hétéroclite.

- Elle désigne également, dans le domaine des interfaces graphiques, une bibliothèque de widgets, c'est-à-dire de composants graphiques de conception et apparence *(look and feel)* homogènes. C'est pour cette raison que dojo a été nommé « dojo toolkit » par ses créateurs.

Une *bibliothèque* est un ensemble de composants, ceux-ci étant soit des fonctions et procédures (bibliothèques de fonctions), très répandues dans le monde scientifique, notamment pour les calculs mathématiques, soit des classes ou encore des objets.

Le plus souvent, chaque bibliothèque est dédiée à un domaine particulier et forme un tout cohérent. En PHP, par exemple, nous avons les bibliothèques CURL, dédiée à l'appel d'URL distantes, ou DOM XML, consacrée à DOM et à XSLT. En Java, J2SE (Java 2 Standard Edition), est un ensemble de bibliothèques, dont jdbc, qui interface Java avec les SGBD, est l'une des plus connues.

Le travail sur la cohérence des bibliothèque peut être très approfondi, en particulier dans le cas des langages objet, chaque bibliothèque fournissant une hiérarchie d'héritage de classes. Le nommage des services fournis par les classes peut alors être rendu homogène, notamment grâce à l'héritage (si une classe hérite d'une autre, elle en hérite en particulier les noms de services). Les bibliothèques d'Eiffel sont sans doute celles dans lesquelles cet effort de cohérence a été poussé le plus loin.

Dans les bibliothèques, la cohérence et l'homogénéité des noms et du style sont importantes en ce qu'elles facilitent la tâche du développeur qui utilise ces bibliothèques, en réduisant le nombre d'informations qu'il doit mémoriser. Le travail de leurs concepteurs est difficile et nécessite qu'ils aient une vision globale de l'ensemble des composants de la bibliothèque, et parfois de plusieurs bibliothéques.

On parle d'API (Application Programming Interface), ou interface de programmation, pour désigner les *spécifications* des composants de ces bibliothèques : signatures des méthodes, pré- et postconditions des méthodes, exceptions qui peuvent être déclenchées, voire invariants de classe.

Un *framework*, ou cadre d'applications, fournit à travers ses API un cadre pour développer un type d'application (par exemple, les applications Web). Il fait un choix sur la façon d'organiser l'ensemble du code, en définissant les composants à produire, ainsi que leur rôle et comment ils interagissent, ce que l'on appelle communément *l'architecture* de l'application, et en fournit la partie qui se retrouve dans toutes les applications de même type.

Grâce à cela, il règle les problèmes récurrents auxquels elles font face, rend homogène l'ensemble de l'application, ainsi que les applications entre elles, induit de bonnes pratiques de conception, et réduit le temps de développement, puisqu'il fournit une partie du code à produire et guide la conception. Il peut, en particulier, définir des classes abstraites que le développeur doit implémenter (par exemple, les classes Action de Struts). Tous ces

avantages s'obtiennent évidemment à la condition qu'il soit mûr et bien conçu, ce qui n'est pas toujours le cas.

Pour les applications Web, les frameworks les plus connus, c'est-à-dire Struts et JSF en J2EE et Ruby on Rails, adoptent tous l'architecture MVC (modèle, vue, contrôleur), le framework .NET étant, quant à lui, très spécifique. L'architecture MVC est aujourd'hui un modèle éprouvé, reconnu et valable. Un autre framework très connu, Hibernate, se consacre au problème de la persistance et de la correspondance objet/relationnel, ce qui est fondamental puisque, aujourd'hui, les bases de données sont majoritairement relationnelles et les langages orientés objet et que la correspondance table/classe n'est pas triviale.

Boîtes à outils, bibliothèques et frameworks visent tous à simplifier le code des applications en réutilisant des briques existantes. Les boîtes à outils, dans leur premier sens, juxtaposent simplement des éléments, tandis que les bibliothèques et les boîtes à outils graphiques visent à les intégrer en un ensemble et que les frameworks vont encore plus loin, en indiquant au développeur comment organiser le code de son application.

Parler de « framework Ajax » semble donc, actuellement (été 2006), quelque peu abusif. Nous disposons en réalité de bibliothèques de composants JavaScript, de taille et d'intégration diverses, et d'additifs côté serveur (bibliothèques PHP, Java, JSP, ou multiplates-formes), qui intègrent avant tout ces composants. Une liste à jour est disponible à l'adresse *http://ajaxpatterns.org/Ajax_Frameworks*.

Les bibliothèques JavaScript

L'offre de bibliothèques JavaScript est extrêmement diverse et mouvante, le marché étant récent et loin d'être stabilisé. La question du choix semble ainsi épineuse. Les critères à prendre en compte sont les fonctionnalités offertes, la pérennité et le support, ainsi que la qualité de la conception et de la documentation.

En ce qui concerne la pérennité, la donne a changé au printemps 2006, quand Yahoo et Google ont publié leurs API. L'une et l'autre vont sans doute devenir très populaires. D'un autre côté, les bibliothèques prototype.js et script.aculo.us sont intégrées au framework Web Ruby on Rails, très populaire aux États-Unis. Enfin, dojo a reçu le soutien d'IBM. Ces bibliothèques sont ainsi assurées d'une certaine pérennité.

L'approche de Google sort du lot. Considérant le développement JavaScript fastidieux et propice aux erreurs, Google propose d'écrire les applications Ajax entièrement en Java, en bénéficiant de l'incomparable outil de développement qu'est Eclipse. Pour cela, Google a mis au point une vaste bibliothèque de classes couvrant tout ce dont le développeur a besoin pour bâtir des applications Web, allant des éléments classiques du HTML (formulaires, boutons, tableaux, liens, etc.) à un ensemble de widgets qui en sont absents (onglets, arbres, menus, etc.), en passant par les appels Ajax et un mécanisme de gestion de l'historique de navigation.

Le GWT (Google Web Toolkit) dispose d'un compilateur qui transforme le code Java en du HTML et JavaScript portable, sachant que, pendant le développement, la mise au point se fait entièrement en Java, et non sur le code généré.

C'est une approche intéressante, et à considérer lorsque nous développons le serveur en Java. Cependant, il faut s'assurer que la mise au point peut effectivement se faire intégralement au niveau Java, ce sur quoi on peut nourrir quelques doutes, la génération de code étant rarement aussi transparente que ses promoteurs veulent bien le laisser croire. Par ailleurs, cette approche va à l'encontre des usages actuels, fondés sur le HTML, et donc des applications existantes dans lesquelles GWT serait peut-être difficile à intégrer.

Les autres bibliothèques citées précédemment sont toutes en JavaScript. Celle de Yahoo est bien documentée, avec des exemples d'utilisation. En revanche, dojo et prototype sont peu ou mal documentées. Les choses s'amélioreront sans doute avec le temps et le soutien d'acteurs de poids, comme IBM pour dojo.

En matière de fonctionnalités, les bibliothèques permettent d'effectuer les opérations suivantes :

- encapsuler les différences des navigateurs concernant le HTML dynamique ;
- fournir des composants graphiques ;
- réduire le code des appels Ajax ;
- étendre certains objets JavaScript prédéfinis.

prototype.js

Écrite par Sam Stephenson, prototype.js est la bibliothèque de base qui est fournie avec Ruby on Rails et sur laquelle plusieurs autres bibliothèques sont fondées, notamment script.aculo.us ou rico, côté client, et Ajax JSP Tags (ensemble de balises JSP), côté serveur.

De taille réduite (47 Ko dans sa version 1.4), elle se charge rapidement. Une documentation non officielle, à l'initiative d'un de ses utilisateurs, est disponible en anglais à l'adresse *http://www.sergiopereira.com/articles/prototype.js.html*. Très peu documenté, le code de prototype.js fait abondamment appel aux particularités de JavaScript exposées au chapitre 3.

Le parti pris est celui de la concision et de l'efficacité. Par exemple, les constructeurs ne vérifient pas la validité des paramètres. Cela allège le code de prototype.js mais rend son utilisation moins facile lors de la mise au point. De même, les objets définis n'ont pas d'espace de noms, ce qui peut poser des problèmes si nous intégrons du code de provenances diverses.

Fonctions et objets utilitaires

La bibliothèque prototype.js propose un certain nombre de raccourcis pour rendre le code plus concis.

Par exemple :

```
$("unDiv").innerHTML = "blabla";
```

est équivalent à :

```
document.getElementById("unDiv").innerHTML = "blabla";
```

En fait, la fonction `$` va plus loin. Elle peut prendre en argument une chaîne, auquel cas elle est strictement équivalente à `document.getElementById`, ou bien une liste d'id, auquel cas elle renvoie un tableau d'éléments DOM.

Nous pouvons ainsi écrire :

```
fields = $("idUser", "motPasse");
```

et récupérer un tableau de deux éléments DOM.

Elle peut aussi prendre en argument un nœud DOM, au lieu de son id, auquel cas elle le renvoie, ce qui est pratique, par exemple pour définir des constructeurs :

```
function UnConstructeur(idRoot) {
  this.root = $(idRoot);
  // etc.
}
```

Nous pouvons appeler ce constructeur soit en passant l'id de l'élément racine, soit en passant directement cet élément, si nous l'avons récupéré par ailleurs. Cela rend plus flexibles les fonctions ayant pour argument des éléments DOM, comme nous allons très vite nous en rendre compte.

Un autre raccourci donne la valeur d'un champ :

```
var nomUser = $F("user"); // "user" est le id du champ
```

`$F(id)` est un raccourci pour `$(id).value`.

Nous pouvons obtenir les champs nommés d'un formulaire sous forme sérialisée, avec la méthode `serialize` de l'objet `Form` défini par prototype.

Par exemple, si nous avons le formulaire suivant :

```
<form id="personne" action="uneAction.php">
  Prénom : <input type="text" name="prenom"/>
  <br/>
  Nom : <input type="text" name="nom"/>
  <br/>
  <input type="button" value="soumettre"/>
</form>
```

et si l'utilisateur saisit « Molière » dans le nom, et rien dans le prénom, le code :

```
var corps = Form.serialize($("personne"));
```

affiche :

```
prenom=&nom=Moli%C3%A8re
```

Nous remarquons que les valeurs produites sont encodées suivant `encodeURIComponent`, et que seuls les champs ayant un nom sont pris en compte, le bouton, qui n'a pas d'attribut `name`, étant ignoré. L'objet `Form` a d'autres méthodes, comme `disable`, qui désactive tous les champs de saisie du formulaire passé en paramètre (nous pouvons passer le formulaire lui-même ou son id).

Comme nous l'avons vu au chapitre 3, prototype.js a sa façon de déclarer des « classes ».

Pour créer un type d'objet `User`, nous écrivons :

```
User = Class.create();
```

Le constructeur est défini comme la méthode `initialize` du prototype de `User` :

```
User.prototype = {
  initialize: function(idLogin, idPassword, idMsg) {
    // etc.
```

Nous disposons aussi de la méthode `extend` de `Object`, qui ajoute à un objet, et en particulier à un prototype, d'autres propriétés.

C'est ainsi que prototype.js étend l'objet `String`, prédéfini dans JavaScript :

```
Object.extend(String.prototype, {
  // Enlever toutes les balises de la chaine courante
  stripTags: function() {
    return this.replace(/<\/?[^>]+>/gi, '');
  },
  // etc.
}
```

Nous pouvons alors écrire :

```
var s = "<p id='test'>blabla</p>";
alert(s.stripTags()) ; // affiche "blabla"
```

Parmi les autres extensions de l'objet String, citons `escapeHTML`, qui transforme les caractéres HTML en leurs entités (`<` devient `<` etc.) et `unescapeHTML`, qui fait l'inverse, et `camelize`, utile en CSS, qui transforme par exemple `border-style` en `borderStyle`.

`Array` est un autre objet natif de JavaScript qu'étend prototype.js.

Plus précisément, il récupère toutes les propriétés de l'objet `Enumerable` défini par prototype.js grâce à la ligne suivante :

```
Object.extend(Array.prototype, Enumerable);
```

Cet objet `Enumerable` porte la marque du langage Ruby. Il permet d'effectuer des traitements sur tous les éléments de l'énumération.

Pour les illustrer, partons de l'objet de type `Array` suivant :

```
var tab = [1, 3, 5, 7, 8, 9, 10, 15, 20];
```

Le tableau 6.1 récapitule les méthodes principales de `Enumerable` en les appliquant à cet objet `tab`. Chaque méthode prend en argument une fonction à appliquer à chaque élément du tableau, fonction qui dépend de deux paramètres : `value`, valeur de l'élément courant, et `index`, rang de cet élément.

Passer en argument une fonction est une des caractéristiques de JavaScript qui rendent ce langage si flexible. La bibliothèque prototype.js en fait là un usage judicieux, qui peut nous économiser beaucoup de code.

Tableau 6.1 Principales méthodes de l'objet *Enumerable* de prototype.js

Méthode	Résultat	Description

`tab.all(function(value, index) {`
`  return value > 0;`
`})`
	`true`	Indique si tous les éléments satisfont la condition testée par la fonction passée en paramètre. Sans paramètre, `all` indique si tous les éléments sont assimilables à `true` (non nuls s'ils sont de type `Number`, différents de `null` s'ils sont de type objet). Ici, la méthode indique s'ils sont tous positifs.

`tab.any(function(value, index) {`
`  return value > 16;`
`})`
	`true`	Indique si un élément au moins vérifie la condition. Sans paramètre, indique si un élément est assimilable à `true` (c'est-à-dire est non `null` si c'est un objet, ou non `nul` si c'est un nombre). Ici, la méthode indique si un est plus grand que 16.

`tab.any(function(value, index) {`
`  return value > 30;`
`})`
	`false`	Indique ici si un élément est plus grand que 30.

`tab.findAll(function(value, index) {`
`  return value > 9;`
`})`
	`[10, 15, 20]`	Renvoie le tableau de tous les éléments vérifiant la condition passée en paramètre sous forme de fonction.
		Ici, la méthode renvoie tous les éléments plus grands que 9.

`tab.find(function(value, index) {`
`  return value > 9;`
`})`
	`10`	Renvoie le premier élément trouvé, ou `null` s'il n'y en a pas.

`tab.max()`	`20`	Renvoie le plus grand élément.

`tab.max(function(value, index) {`
`  return 30 - value;`
`})`
	`29`	Renvoie la plus grande valeur calculée par la fonction passée en paramètre. Ici, la méthode renvoie le plus grand écart à 30.

`tab.include(8) ;`	`true`	Indique si l'élément passé en paramétre est présent dans la collection.

Pouvoir passer une fonction à la méthode `max` est utile lorsque les valeurs ne sont pas numériques, par exemple si le tableau est un tableau d'éléments DOM, et que nous comparons le nombre d'enfants de ces éléments.

Les composants que nous avons construits au cours des chapitres précédents ont le plus souvent parmi leurs attributs des éléments DOM, dont le composant doit spécifier les réactions. Lorsque celles-ci doivent faire appel à une méthode du composant, nous devions alors mémoriser le composant dans une variable. prototype.js permet de nous affranchir de cette contrainte.

Considérons un composant très simple, qui affiche dans une zone de message le nombre de fois que l'utilisateur a cliqué sur un élément donné. Sans prototype.js, nous écririons :

```
function ClickCounter(element, msg) {
  this.element = document.getElementById(element);
  this.msg = document.getElementById(msg);
  this.nbClicks = 0;
  var current = this;←❶
  this.element.onclick = function(event) {
    current.onclick(event);←❷
  }
}
ClickCounter.prototype.onclick = function(event) {
  this.nbClicks++;
  this.msg.innerHTML = "Zone cliquée " + this.nbClicks + " fois";
}
new ClickCounter("test", "msg");
```

En ❶, nous mémorisons l'objet courant, que nous utilisons en ❷.

Avec prototype.js, le code du constructeur devient :

```
function ClickCounter(element, msg) {
  this.element = $(element);
  this.msg = $(msg);
  this.nbClicks = 0;
  this.element.onclick = this.onclick.bindAsEventListener(this);←❶
}
```

Nous n'avons plus besoin de mémoriser l'objet courant, grâce à la méthode `bindAsEventListener`, ajoutée par prototype.js à `Function.prototype`, et ainsi disponible dans toute fonction. Cette méthode renvoie une fonction qui applique à l'objet passé en paramètre (ici, `this`) la méthode sur laquelle elle est appelée (ici, `this.onclick`), avec pour paramètre l'événement courant.

La ligne ❶ est ainsi strictement équivalente à l'écriture initiale :

```
var current = this;
this.element.onclick = function(event) {
  current.onclick(event);
}
```

Cette fonction très pratique est abondamment utilisée dans prototype.js et pour construire des composants.

Encapsulation et extensions de DOM

De la même façon que nous l'avons fait nous-mêmes tout au long des chapitres précédents, prototype.js encapsule les divergences des navigateurs concernant DOM et ajoute des fonctions au travers de deux objets : `Element`, qui regroupe les méthodes concernant les éléments, et `Event`, consacré aux événements.

Le tableau 6.2 recense les méthodes phares de `Element`. Toutes prennent pour argument un élément DOM, soit l'objet lui-même, soit son id (elles utilisent toutes la fonction `$` vue précédemment).

Tableau 6.2 Principales méthodes de l'objet *Element* de prototype.js

Méthode	Effet
`cleanWhitespaces(element)`	Supprime les nœuds texte vides enfants de `element`.
`empty(element)`	Indique si `element` est vide ou ne contient que des espaces.
`getDimensions(element)`	Renvoie les dimensions de `element` sous la forme d'un objet à deux propriétés : `width` et `height`.
`makePositioned(element)`	Met la directive CSS `position` de `element` à `"relative"`. Indispensable pour le glisser-déposer.
`remove(element)`	Enlève l'élément `element` de l'arbre DOM.
`scrollTo(element)`	Fait défiler la fenêtre jusqu'à la position de `element`.
`undoPositioned(element)`	Remet la directive CSS `position` de `element` à vide. L'élément revient dans le flot normal.

Le tableau 6.3 est l'analogue pour l'objet `Event` du tableau précédent. Les méthodes prennent toutes pour argument l'événement courant. Il s'y ajoute une série de constantes.

Tableau 6.3 Principales propriétés de l'objet *Event* de prototype.js

Méthode	Effet
`element(event)`	Élément source de l'événement (`event.target` ou `event.srcElement`, selon le navigateur)
`isLeftClick(event)`	Indique si le bouton gauche de la souris est enfoncé.
`pointerX(event)` `pointerY(event)`	Retourne les coordonnées de la souris sur la page.
`stop(event)`	Bloque le comportement par défaut et la propagation de l'événement.
`findElement(event, tagName)`	Retourne le premier ancêtre de la source de l'événement de type `tagName`, ou `null` si rien n'est trouvé.

`KEY_BACKSPACE: 8`
`KEY_TAB: 9`
`KEY_RETURN: 13`
`KEY_ESC: 27`
`KEY_LEFT: 37`
`KEY_UP: 38`
`KEY_RIGHT: 39`
`KEY_DOWN: 40`
`KEY_DELETE: 46`
	Noms des touches du clavier, correspondant à un numéro Unicode

En plus de cela, prototype.js propose deux objets, `Insertion` et `Position`, ce dernier aidant à positionner des éléments. `Insertion` insère du contenu par rapport à un élément et se décline en quatre variantes, comme indiqué au tableau 6.4.

Tableau 6.4 Objets *Insertion* de prototype.js

Instruction	Effet
`Insertion.Before(element, contenu)`	Insère `contenu` juste avant `element`.
`Insertion.Top(element, contenu)`	Insère `contenu` juste au début du contenu de `element`. Il en devient le premier nœud texte enfant.
`Insertion.Bottom(element, contenu)`	Insère `contenu` juste à la fin du contenu de `element`. Il en devient le dernier nœud texte enfant.
`Insertion.After(element, contenu)`	Insère `contenu` juste après `element`.

Dans l'exemple suivant illustrant ces instructions, prenons l'élément HTML :

```
<div id="texte">blabla</div>
```

et observons les effets d'une insertion, récapitulés sur le tableau 6.5.

Tableau 6.5 Utilisation des objets *Insertion* de prototype.js

Instruction	Résultat
`new Insertion.Before("texte", "<hr/>")`	`<hr/><div id="texte">blabla</div>`
`new Insertion.Top("texte", "<hr/>")`	`<div id="texte"><hr/>blabla</div>`
`new Insertion.Bottom("texte", "<hr/>")`	`<div id="texte">blabla<hr/></div>`
`new Insertion.After("texte", "<hr/>")`	`<div id="texte">blabla</div><hr/>`

Les appels XMLHttpRequest

Les principaux attributs et méthodes de `XMLHttpRequest` *(voir en annexe pour une description détaillée de ces attributs et méthodes)* sont portables sur tous les navigateurs. L'instanciation de cet objet peut en outre être rendue identique dans tous les navigateurs en étendant l'objet `window`, comme nous l'avons vu au chapitre 4.

La seule chose un peu fastidieuse consiste à répéter dans le code de `onreadystatechange` le test :

```
if (request.readyState == 4 && request.status == 200)
```

La bibliothèque prototype.js nous en affranchit en proposant un objet `Ajax.Request`, doté de méthodes `onSuccess`, `onLoading`, etc.

Nous lançons une requête `XMLHttpRequest` à une URL par la ligne :

```
ajaxCall = new Ajax.Request(url, options);
```

dans laquelle `options` est un objet (optionnel) pouvant prendre les propriétés indiquées au tableau 6.6.

Tableau 6.6 Options des appels Ajax de prototype.js

Option	Description
`method`	Méthode (par défaut `post`)
`parameters`	Chaîne de paramètres à transmettre (par défaut, chaîne vide)
`asynchronous`	Booléen indiquant si la requête est asynchrone ou non (par défaut, `true`).
`postBody`	Corps éventuel. Il est inutile de renseigner cette option si `parameters` est renseigné.
`requestHeaders`	Liste d'en-têtes à passer à la requête. C'est un tableau de chaînes allant par deux : d'abord le nom de l'en-tête, ensuite sa valeur.
`onLoading`	Fonction à exécuter quand la requête est envoyée. Prend cet objet requête en paramètre.
`onComplete`	Fonction à exécuter quand la réponse est arrivée (`readyState == 4`). Prend la requête `XMLHttpRequest` sous-jacente en paramètre.
`onSuccess`	Fonction à exécuter quand la réponse est arrivée et que le statut indique le succès (valeurs dans la plage 200-299) ou est nul (lorsque la requête a été annulée) ou indéfini. Prend la requête `XMLHttpRequest` sous-jacente en paramètre.
`onFailure`	Fonction à exécuter quand la réponse est arrivée mais sans succès, c'est-à-dire que le statut ne remplit pas les conditions de `onSuccess`. Prend la requête `XMLHttpRequest` sous-jacente en paramètre.
`onException`	Fonction à exécuter s'il s'est produit une exception. Prend la requête `XMLHttpRequest` sous-jacente en paramètre.

La requête `XMLHttpRequest` sous-jacente s'obtient par la propriété `transport` de `Ajax.Request`.

Nous allons utiliser cet objet `Ajax.Request` pour faire une requête simple, qui ira chercher l'heure du serveur.

Comme au chapitre 1, nous prévoyons un bouton pour lancer la requête, un pour l'annuler et une zone pour afficher le résultat ainsi que le message indiquant que la requête est en cours :

```
<body>
  <h1>Récupérer l'heure du serveur avec prototype.js</h1>
  <form action="javascript:;">
    <input type="submit" id="runRequest" value="Lancer"/>
    <input type="button" id="stopRequest" value="Arrêter"/>
  </form>
  <div id="msg"></div>
```

Nous spécifions ensuite le comportement :

```
var ajaxCall;←❶
$("runRequest").onclick = function() {
  ajaxCall = new Ajax.Request("serveur/get-time.php", {
    method: "get",
    onLoading: function() {←❷
```

```
      $("msg").innerHTML = "En chargement...";
    },
    onSuccess: function(request) {←❸
      $("msg").innerHTML = request.responseText;
    },
    onFailure: function(request) {←❹
      $("msg").innerHTML = "Erreur " + request.status;
    }
  });
}
```

En ❶, nous déclarons l'objet de type `Ajax.Request`. En ❷, nous spécifions l'action à effectuer quand la requête est lancée, en l'occurrence en avertir l'utilisateur, comme l'illustre la figure 6.1. En ❸ vient la réaction lorsque la réponse est parvenue sans problème : nous affichons dans la zone de message le texte de la réponse obtenue à partir de la requête, passée en paramètre. Enfin, en ❹, si un problème a surgi, nous indiquons le statut de la réponse.

Dans toutes ces méthodes, nous utilisons abondamment la fonction `$`. L'écriture en apparence événementielle apporte une netteté indéniable au code.

Pour annuler la requête, rien n'est proposé par prototype.js. Le code revient donc à ce que nous écrivions aux chapitres 4 et 5 :

```
$("stopRequest").onclick = function() {
  if (ajaxCall && ajaxCall.transport &&
      ajaxCall.transport.readyState != 4) {
    ajaxCall.transport.abort();
    $("msg").innerHTML = "Abandon";
  }
}
```

Nous annulons la requête `XMLHttpRequest`, obtenue par `ajaxCall.transport`, dans le cas où il y en a bien une et qu'elle n'est pas déjà terminée.

Il est possible avec `Ajax.Request` d'exécuter du code JavaScript envoyé par le serveur si la réponse a pour type MIME `text/javascript`.

Par exemple, si la page serveur **produire-javascript.php** contient :

```
<?php
header("Content-type: text/javascript");
print "alert('ok')";
?>
```

l'appel :

```
new Ajax.Request("serveur/produire-javascript.php");
```

récupère le code JavaScript et l'exécute, faisant apparaître une fenêtre pop-up affichant « ok ».

Deux extensions de `Ajax.Request` sont proposées par prototype.js : `Ajax.Updater` et `Ajax.PeriodicalUpdater`. Toutes deux supposent que la réponse produite est au format

HTML (ou simplement textuel) et remplacent le contenu d'une zone par cette réponse. Bien entendu, cette zone est un paramètre supplémentaire du constructeur.

Par exemple :

```
new Ajax.Updater("meteo", "serveur/meteo.php");
```

met à jour l'élément HTML d'id `meteo`, en remplaçant son contenu par la réponse de **meteo.php.**

L'objet `Ajax.PeriodicalUpdater` est utile pour afficher un fragment HTML toujours à jour, par exemple la météo, un flux boursier, un trafic réseau ou toute autre donnée de supervision, pour peu qu'elle soit disponible sur le réseau sous forme HTML.

Voici comment adapter l'exemple très simple ci-dessus, qui affiche l'heure du serveur (pour qu'il ait quelque intérêt, imaginons qu'il s'agit d'une cotation boursière ou de l'état d'un stock) :

```
var ajaxCall;
$("runRequest").onclick = function() {
  ajaxCall = new Ajax.PeriodicalUpdater($("msg"),
    "serveur/get-time.php", { method: "get", frequency: 1 });
}
```

Comme il n'est pas très utile, dans ce cas précis, de montrer que le navigateur effectue des requêtes au serveur, nous supprimons l'option `onLoading`. Une nouvelle option, `frequency`, indique le nombre de secondes à laisser s'écouler entre deux requêtes successives. Par défaut, ce nombre vaut 2.

Pour arrêter la mise à jour périodique, `Ajax.PeriodicalUpdater` dispose de la méthode `stop` :

```
$("stopRequest").onclick = function() {
  if (ajaxCall) {
    ajaxCall.stop();
  }
}
```

Nous avons parcouru l'essentiel de la bibliothèque prototype.js. Nous y avons trouvé des constructions de base pratiques, facilitant l'écriture d'un code concis, ce qui explique sans doute qu'elle serve de soubassement à d'autres bibliothèques, comme script.aculo.us ou rico.

script.aculo.us

Cette bibliothèque, également intégrée à Ruby on Rails, propose quelques widgets, notamment `Slider`, une règle réagissant au mouvement du curseur, un support de glisser-déposer, quelques effets, par exemple pour changer l'opacité d'un élément ou le déplacer, et un composant de suggestion de saisie. Elle est téléchargeable sur le site *http://script.aculo.us*, qui propose en outre de la documentation et des exemples de code.

Pour l'utiliser, il faut inclure dans la page prototype.js puis scriptaculous.js. Le code de script.aculo.us est réparti en plusieurs fichiers de petite taille (moins de 30 Ko), chacun étant dédié à un sujet : glisser-déposer, widgets, effets. Il est possible de n'inclure que les fichiers nécessaires.

Par exemple, pour utiliser les effets et le glisser-déposer, nous écrivons :

```
<head>
  <script src="prototype.js"></script>
  <script src="scriptaculous.js?load=effects,dragdrop"></script>
</head>
```

Cela présente l'avantage de limiter la taille du code à charger, et donc de réduire le temps de réponse. Ces fichiers peuvent se mettre dans le cache du navigateur, ce qui réduit le temps de chargement (mais pas d'interprétation).

Pour illustrer les effets, créons une page avec une zone pourvue d'un style (omis ici) et deux boutons :

```
<head>
  <script src="prototype.js"></script>
  <script src="scriptaculous.js?load=effects"></script>
  <!-- etc -->
</head>
<body>
  <div id="zone">Une zone de contenu</div>
  <input type="button" id="opacity" value="Opacité"/>
  <input type="button" id="moveBy" value="Bouger"/>
```

dont l'action fait appel aux effets script.aculo.us suivants :

```
  <script>
$("opacity").onclick = function() {❶
  new Effect.Opacity("zone", { duration: 2, from: 1, to: 0.3 });
}
$("moveBy").onclick = function() {❷
  new Effect.Move("zone", { x: 100, y: 50, mode: "relative"});
}
  </script>
```

Tous les effets se déclarent de la même façon, en prenant en paramètre l'id de l'élément HTML auquel ils s'appliquent et un objet regroupant les options de l'effet, de la même façon que `Ajax.Request` regroupe les options de l'appel `XMLHttpRequest`. Les options dépendent de chaque effet. Ainsi, en ❶, le changement d'opacité de l'élément dure 2 secondes et va de l'opacité maximale (1) à 0.3, tandis qu'en ❷, l'élément est déplacé, d'un mouvement continu, de 100 pixels vers la droite et 50 vers le bas en position relative.

La figure 6.1 illustre le résultat avant et après l'application de ces deux effets. Sur la fenêtre de droite, la zone déplacée a un fond moins foncé et qui laisse transparaître le bouton situé sous elle : c'est précisément en cela que consiste une opacité plus faible (ici elle vaut 0.3). Ce changement d'opacité est assez utilisé dans les applications Web 2.0,

certains l'utilisant pour indiquer que l'élément va être ou vient d'être mis à jour par un appel Ajax.

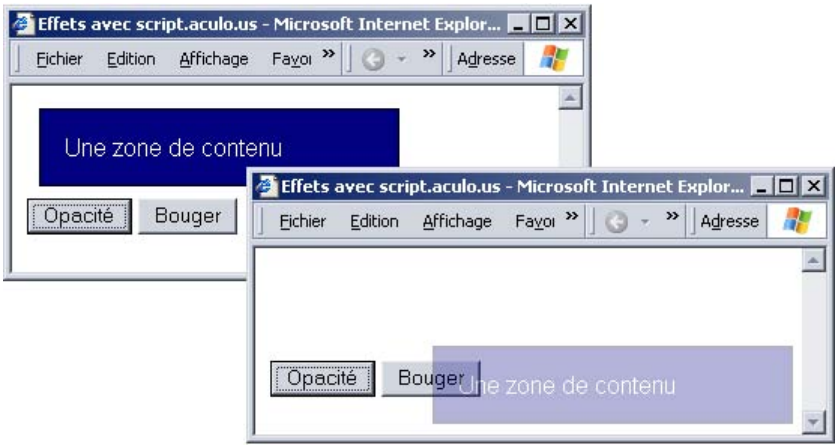


Figure 6.1
Un élément avant et après application de deux effets

rico

La bibliothèque rico *(http://openrico.org)* est elle aussi bâtie sur prototype.js. Sa version 1 a été développée chez Sabre Airline Solutions, qui a vendu à la SNCF, voici quelques années, le logiciel ayant servi de base au célèbre projet Socrate, et elle est maintenant Open Source (licence Apache).

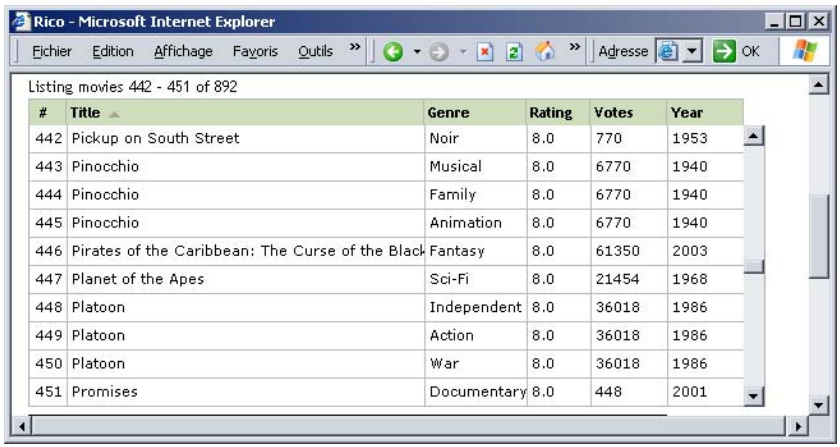
Elle se présente sous la forme d'un bloc unique, d'une taille moyenne (90 Ko). En espagnol, rico signifie riche, ce qui suggère ici « client riche ».

La bibliothèque rico propose un mécanisme de glisser-déposer, ainsi qu'un objet permettant de manipuler les couleurs (passer de RGB à HSB, etc.), des effets d'arrondis pour les boîtes, d'autres effets similaires à ceux de script.aculo.us, des accordéons et des « datagrid », c'est-à-dire des tableaux évolués permettant d'afficher un grand nombre d'enregistrements, et dont on peut trier les colonnes.

À tout moment, seule une partie de l'ensemble est disponible sur le client, les parties adjacentes étant récupérées par des appels Ajax quand l'utilisateur fait défiler la barre de défilement.

La figure 6.2 illustre ce composant. Notons la petite icône indiquant dans la cellule de titre « Title » que les données sont triées par titre croissant. Lorsque l'utilisateur clique sur un titre de colonne ou fait défiler le tableau au-delà de ce qui est stocké sur le client, une requête Ajax est émise vers le serveur et met à jour le tableau. C'est une utilisation intelligente d'Ajax, qui permet d'avoir en mode Web un comportement similaire à celui des clients lourds.

Ce composant comporte cependant un défaut de conception : les requêtes sont envoyées sans que le composant annule les requêtes émises précédemment et encore en cours, si bien que nous pouvons nous retrouver avec des requêtes parallèles, ce qui surcharge inutilement l'interprète JavaScript et le réseau et peut conduire le navigateur à se figer, comme nous l'avons vu au chapitre 4.



#	Title	Genre	Rating	Votes	Year
442	Pickup on South Street	Noir	8.0	770	1953
443	Pinocchio	Musical	8.0	6770	1940
444	Pinocchio	Family	8.0	6770	1940
445	Pinocchio	Animation	8.0	6770	1940
446	Pirates of the Caribbean: The Curse of the Black	Fantasy	8.0	61350	2003
447	Planet of the Apes	Sci-Fi	8.0	21454	1968
448	Platoon	Independent	8.0	36018	1986
449	Platoon	Action	8.0	36018	1986
450	Platoon	War	8.0	36018	1986
451	Promises	Documentary	8.0	448	2001

Figure 6.2
Datagrid de rico, trié par titre croissant

Le glisser-déposer se code simplement. Nous allons réaliser l'exemple illustré à la figure 6.3, dans lequel l'utilisateur fait passer une zone d'un conteneur à un autre. Remarquons que, pendant le déplacement, le fond des deux conteneurs change de couleur, avant de revenir à l'état initial une fois la zone relâchée sur le deuxième conteneur. C'est la façon qu'a rico de montrer à l'utilisateur les zones où il peut déposer la zone déplaçable.

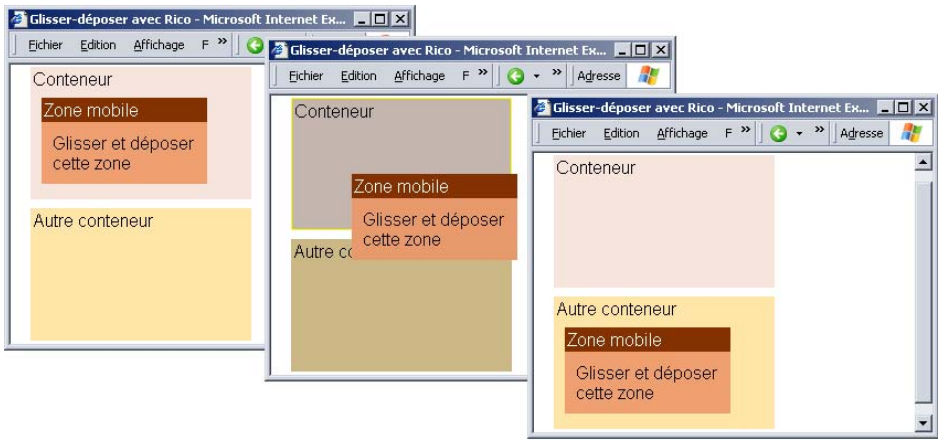


Figure 6.3
Glisser-déposer avec rico (avant, pendant et après)

Dans la page HTML, nous incluons les bibliothèques nécessaires (prototype et rico) :

```
<script src="prototype.js"></script>
<script src="rico.js"></script>
```

puis écrivons le corps de la page :

```
<body>
  <div id="conteneur1" class="panel" style="background:#F7E4DD">
    Conteneur
    <div id="zoneMobile" class="box" style="background:#f0a070">
      <div class="boxTitle">Zone mobile</div>
      <div>Glisser et déposer cette zone</div>
    </div>
  </div>
  <div id="conteneur2" class="panel" style="background:#FFE5A8">
    Autre conteneur
  </div>
```

Nous avons simplement trois `div`, avec quelques classes CSS pour la stylisation, et des id pour notre code JavaScript, que voici :

```
dndMgr.registerDraggable( new Rico.Draggable("","zoneMobile") );←❶
dndMgr.registerDropZone( new Rico.Dropzone("conteneur1") );←❷
dndMgr.registerDropZone( new Rico.Dropzone("conteneur2") );←❸
```

En ❶, nous faisons de l'élément d'id `zoneMobile` une zone déplaçable et l'enregistrons dans le gestionnaire de glisser-déposer. En ❷ et ❸, nous faisons des éléments `conteneur1` et `conteneur2` des zones où l'utilisateur peut déposer un élément. L'élément `zoneMobile` peut alors être déposé sur l'une de ces zones.

La bibliothèque propose des mécanismes plus élaborés de glisser-déposer, permettant, par exemple, de limiter, pour chaque zone mobile, l'ensemble des éléments sur lesquels elle peut être déposée.

dojo

Si prototype.js joue la carte *small is beautiful*, dojo *(http://dojotoolkit.org)* pencherait plutôt pour *whole is beautiful.* En effet, ce n'est plus une bibliothèque, mais une véritable plate-forme de développement client que propose dojo, qui n'est pas sans évoquer J2SE, l'API standard de Java, dont il reprend parfois les noms de classes et de méthodes.

Sans nul doute le plus grand ensemble intégré de bibliothèques JavaScript, dojo couvre un champ immense : extensions du langage et des objets JavaScript natifs, encapsulation et extensions de DOM, ensemble considérable de widgets (panneaux, menus, accordéons, onglets, arbres, fenêtres pop-up, etc.), effets visuels, glisser-déposer, support d'Ajax et même de SVG. Il offre en outre également un mécanisme de compression de code, déjà mentionné au chapitre 3, ainsi que des facilités de développement (debug, test unitaire).