5

Échange de données en XML ou JSON

Les deux chapitres précédents, consacrés à JavaScript et à la communication asynchrone avec le serveur grâce à l'objet `XMLHttpRequest`, ont traité ce que recouvrent les trois premières lettres de l'acronyme AJAX (Asynchronous JavaScript and XML). Ce chapitre est consacré à ce que recouvre la derniére, à savoir XML.

XML (eXtensible Markup Language), ou langage de balisage extensible, a été conçu en 1998 afin de permettre d'échanger des *informations structurées portables* sur Internet. C'est une forme simplifiée du SGML (Standard Generalized Markup Language), ou langage normalisé de balisage généralisé.

Les concepteurs de XML avaient deux objectifs en vue :

- Les applications documentaires, en particulier ce qu'on appelle aujourd'hui le « Web sémantique ». XML permet de définir la structure des documents, en contraste avec le HTML, qui en définit surtout l'apparence. Les flux RSS, devenus le standard de fait pour la diffusion d'actualités sur Internet, en sont le résultat le plus probant. Les fichiers XML sont vus dans ce cas comme des *documents*.
- L'EDI (échange de données informatisées) et les services Web, c'est-à-dire la communication entre applications distantes, *via* HTTP. Le résultat le plus connu est sans doute le protocole SOAP (Simple Object Access Protocol), qui définit la structure des données XML échangées par les services Web. Les fichiers XML sont vus dans ce cas comme des *données*.

En fait, XML est avant tout devenu omniprésent dans un troisième domaine : les fichiers de configuration. C'est aussi un format de stockage de plus en plus utilisé par les applica-

tions. À titre d'exemple, les playlists de Windows Media Player sont stockées sous forme de fichiers XML. Par contre, XML s'est diffusé plus lentement dans les applications documentaires ou l'EDI, pour lesquels il a pourtant été conçu.

Ajax et le Web 2.0 sont en train de changer cette situation. Du côté des applications centrées sur les documents, des sites entiers sont bâtis autour de RSS et d'Ajax (par exemple, *www.netvibes.com*). Pour les applications centrées sur les données, l'échange de données entre le serveur et le client par `XMLHttpRequest` révèle au grand jour le potentiel de XML et des technologies liées, notamment XSLT et XPath, ces dernières étant disponibles dans IE et Mozilla, mais pas encore dans Opera ou Safari.

Nous étudions dans ce chapitre l'ensemble de ces possibilités. Après un point succinct sur la syntaxe XML et les différences entre le DOM XML et le DOM HTML, nous examinons les possibilités de XSLT et comparerons les divers formats d'échange entre le client et le serveur : texte simple, HTML, XML, mais aussi JSON, qui se présente comme une solution concurrente de XML.

Au chapitre précédent, nos appels Ajax renvoyaient du texte simple. Ici, ils renverront du XML.

Nous illustrons ce chapitre par deux exemples :

- la mise à jour d'une liste déroulante, portée en XML (application centrée sur les données) ;
- un composant affichant un flux RSS, si important dans le Web 2.0 (application centrée sur les documents).

Nous réalisons en outre quelques manipulations de fichiers de paramètres, puisque XML trouve dans ce domaine son terrain de prédilection.

XML (eXtensible Markup Language)

Ce que nous avons dit au chapitre 2 sur le xHTML s'applique aussi à XML : un document XML peut être vu comme une imbrication de balises sur le disque ou sur le réseau, un arbre DOM en mémoire ou des boîtes imbriquées à l'écran. Il est possible d'appliquer des CSS aux documents XML, mais cela n'est pas dans notre propos.

Nous nous penchons dans cette section sur la syntaxe XML et sur son DOM, qui est un peu différent de celui du HTML.

Documents bien formés

Nous avons rappelé au chapitre 2 les conditions pour qu'un document XML soit *bien formé*. La figure 5.1 les rappelle de façon synthétique.

Il existe deux notions distinctes en XML :

- Les documents *bien formés,* qui suivent les régles de syntaxe de XML. Un document bien formé peut être analysé avec succès et produire un arbre DOM en mémoire.

- Les documents *valides,* qui sont non seulement bien formés mais aussi conformes à une DTD (Document Type Definition) ou à un schéma XML, qui en spécifient la structure.

Les types de documents sont fondamentaux en EDI. Les différents acteurs qui échangent leurs données sous forme de documents XML doivent impérativement s'accorder au préalable sur la structure de ces documents.

Parfois, plusieurs types de documents coexistent pour un même domaine. C'est notamment le cas des RSS, dont plusieurs versions se sont succédé et continuent de coexister, ce qui alourdit la tâche des logiciels lecteurs de RSS.

Pour faire de l'Ajax, nous n'avons pas à écrire des DTD ou des schémas XML. Il peut être utile de savoir les lire, mais nous n'en avons pas besoin dans le contexte de cet ouvrage. Nous renvoyons le lecteur intéressé aux ouvrages de référence sur le sujet.

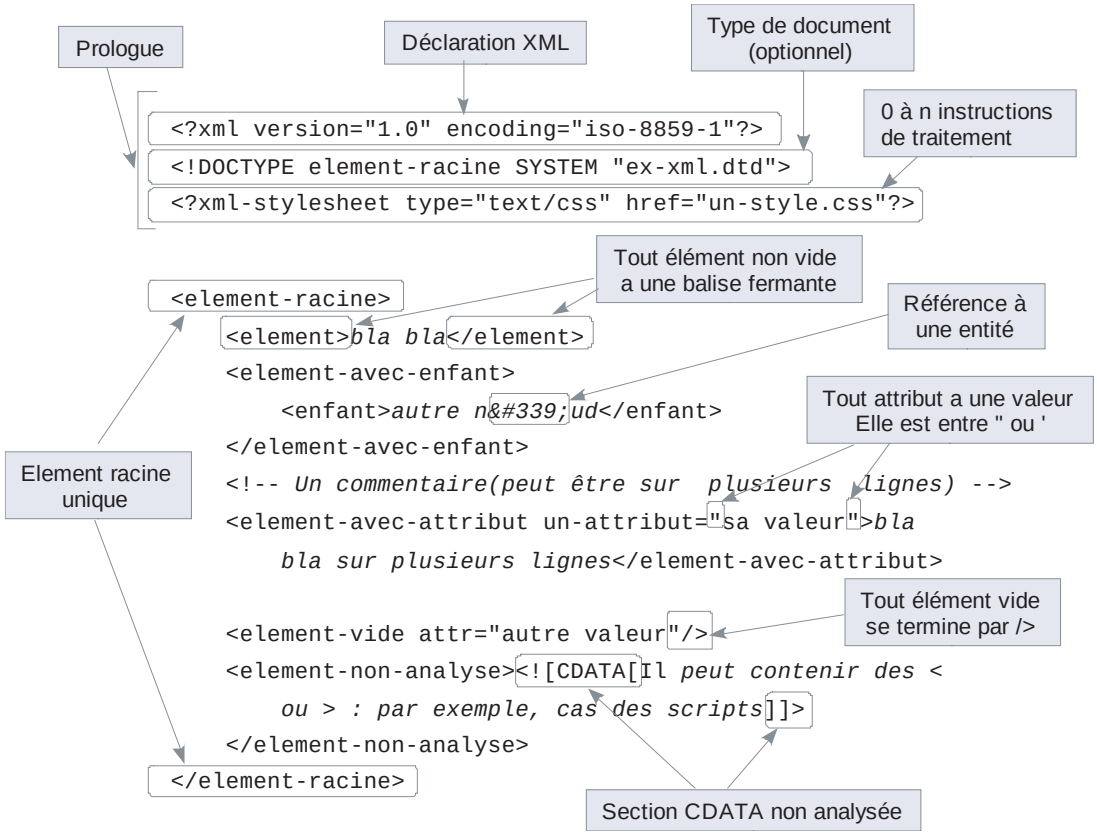


Figure 5.1
Synthèse de la syntaxe XML

Les noms de balises et d'attributs doivent respecter les conventions suivantes :

- commencer par une lettre ou un souligné (caractère « _ ») ;
- continuer par des caractéres valides : lettre, chiffre, deux-points (« : »), point (« . »), moins (« - ») ou souligné (« _ ») ;
- ne pas commencer par « xml », « XML » ou toute variante de ces trois lettres (utilisées dans des mots réservés).

Le caractère « : » est réservé. C'est un délimiteur pour les espaces de noms, que nous traiterons lors de la présentation de XSLT.

Il est possible d'utiliser des caractéres non ASCII, comme les caractéres accentués. Il est toutefois fortement recommandé de se limiter à l'ASCII, disponible sur les claviers du monde entier, qui évite tous les problèmes d'encodage.

Détaillons les points illustrés à la figure 5.1 :

- Le fichier commence obligatoirement par une ligne de déclaration, encadrée par `<?xml` et `?>`. Il est impératif que le fichier commence par la chaîne `<?xml`. S'il y a une espace ou une ligne vide avant ces caractéres ou entre eux, l'analyseur génére une erreur.

  Parmi les attributs de la déclaration, citons notamment les suivants :

  – La version, qui est obligatoire et vaut jusqu'à présent toujours 1.0.

  – Le jeu de caractéres du document, qui est optionnel et doit figurer après la version. En son absence, le document est considéré comme encodé en UTF-8 ou UTF-16 (l'analyseur XML sait deviner).

- Après la déclaration, peuvent venir les éléments suivants :

  – Instructions de traitements, qui sont aussi entre `<?` et `?>`. Seules les instructions de type feuille de style sont implémentées actuellement.

  – Déclaration de type de document, ou DTD (Document Type Definition). Parmi les types bien connus, citons xHTML, RSS, XSLT, XML Schema, MathML et SVG. Les types de documents spécifient la structure des instances de documents. Ils sont l'équivalent des classes dans le monde des documents.

- Le fichier doit contenir ensuite un élément, dit *élément racine*, unique. Cet élément est obligatoire.

Tous les éléments suivent les régles suivantes :

- Tout élément peut être vide ou avoir du contenu ou des éléments enfants. S'il est vide, il est réduit à une balise ouvrante terminée par `/>` au lieu de `>`.
- Il peut avoir un nombre quelconque d'attributs, dont l'ordre n'est pas significatif. Tous les attributs doivent avoir une valeur, entre doubles ou simples cotes.
- Les noms de balises et d'attributs sont sensibles à la casse.

- Il doit y avoir une espace au moins (blanc, tabulation, fin de ligne ou retour chariot) entre le nom de la balise et le nom du premier attribut et entre la valeur d'un attribut et le nom du suivant.

Comme nous l'avons signalé au chapitre 2, il convient de bien distinguer *élément racine* et *racine du document,* cette derniére comportant le prologue et l'élément racine.

Les caractéres `>`, `<`, et `&` sont interdits dans le contenu d'un élément ou la valeur d'un attribut. Il faut les y remplacer par les entités suivantes :

- `<` pour `<` (abréviation de *less than,* ou plus petit que) ;
- `>` pour `>` (abréviation de *greater than,* ou plus grand que) ;
- `&` pour `&` (abréviation d'*ampersand,* ou esperluette).

Au besoin, nous pouvons utiliser `&apos;` (apostrophe) pour le caractère simple cote ('), et `&quot;` (abréviation de *quotation,* ou guillemets) pour les doubles cotes (") dans le contenu des éléments ou la valeur des attributs.

Les commentaires sont délimités par `<!--` et `-->`. Ils ne doivent pas contenir `--` et sont interdits à l'intérieur d'une balise.

Les lignes suivantes sont rejetées par les analyseurs XML :

```
<!-- ceci est un commentaire -- incorrect -->
<item <!-- commentaire interdit ici --> >
```

Scripts et sections CDATA

Si nous produisons par XSLT un document xHTML, nous pouvons être amenés à y inclure du code JavaScript. Or celui-ci comporte fréquemment les caractéres `<`, `>` et surtout `&`, interdits dans le contenu des éléments.

Au lieu d'écrire :

```
if (uneCondition && uneAutreCondition)
```

qui est incorrect à cause des deux esperluettes, nous devrions écrire :

```
if (uneCondition &amp;&amp; uneAutreCondition)
```

ce qui serait illisible.

Pour éviter ce problème, XML définit les sections CDATA (Character DATA), qui sont des éléments que l'analyseur doit passer en mémoire sans les analyser.

Nous pouvons ainsi écrire :

```
<script>
//<![CDTATA[
if (uneCondition && uneAutreCondition) {←❶
  // code javascript
}
//]]></script>
```

Bien que portant un caractère non autorisé (le signe `&`), la ligne ❶ ne pose pas de problème à l'analyseur, car elle se trouve dans une section CDATA. Remarquons que, par sécurité, nous mettons en commentaire les `<![CDTATA[` et `]]` afin d'éviter une erreur JavaScript..

Il est interdit d'inclure la chaîne `]]>` dans une section CDATA, car cette séquence signale la fin de la section. Autrement dit, une section CDATA ne peut en contenir une autre.

Choix de la structure des données

Nous pouvons structurer de diverses manières les flux XML que nous transmettons du serveur au client. Les flux représentant des données sont généralement traités différemment de ceux représentant des documents.

Choix entre attributs et éléments

Les flux représentant des documents suivent une règle simple : les informations destinées à l'utilisateur doivent être placées dans le contenu des éléments, tandis que celles destinées à l'application doivent être placées dans les attributs.

Si nous supprimons toutes les balises, l'utilisateur ne perd aucune information. C'est le cas en xHTML : les attributs servent à la mise en forme (attributs `style`, `class`, `cellpadding`, etc.) ou au travail du navigateur (par exemple, les attributs `href` des liens ou `action` des formulaires lui indiquent l'URL des cibles). Si nous supprimons les balises, nous obtenons un document sans mise en forme, sans images et sans réaction aux liens ou formulaires, mais dont tout le contenu textuel est présent. Les flux RSS étant des documents, ils suivent eux aussi cette régle.

En revanche, la règle ne s'applique pas aux flux de données, pour lesquels elle n'a aucun sens : une suite de données n'est pas un texte et ne peut être comprise sans indications sur la nature de ces données. Une seule règle, de bon sens, s'applique alors : les données composées doivent être des éléments et ne peuvent être des attributs de l'élément auquel elles se rattachent.

Par exemple, considérons la liste des communes d'un code postal donné. L'élément racine, qui pourrait s'appeler `liste`, ne peut avoir les communes trouvées pour attributs, puisque celles-ci sont déjà composées d'un numéro de commune et d'un nom. Elles sont ainsi forcément des éléments, enfants de l'élément `liste`.

La question qui se pose est dés lors de choisir comment représenter les composantes simples d'un élément, c'est-à-dire celles qui n'ont pas de composantes, comme la commune dans notre exemple. Dans le cas où une composante *pourrait* devenir composée ultérieurement, il est envisageable d'en faire un élément. Dans les autres cas, il est généralement préférable d'en faire un attribut, et ce pour deux raisons : ils sont plus faciles à manipuler en DOM (l'accès est direct à travers `getAttribute` et `setAttribute`), et ils réduisent le volume des transferts.

Écrire :

```
<uneBalise uneInfo="blablabla"/>
```

est en effet plus bref qu'écrire :

```
<uneBalise><uneInfo>blablabla</uneInfo></uneBalise>
```

Lorsqu'un flux contient un grand nombre d'éléments, la différence est notable et se traduit dans les temps de réponse de l'ensemble du réseau.

Choix des identifiants

L'un des atouts de XML étant d'être lisible par des êtres humains, il est préférable de privilégier des noms de balises et d'attributs explicites. Le résultat transmis est en quelque sorte autodocumenté, ce qui peut être utile aux développeurs souhaitant l'intégrer dans leurs applications.

Inversement, si les données sont volumineuses, le surpoids causé par les noms de balises et d'attributs doit être pris en compte et réduit autant que faire se peut. Le HTML peut nous servir ici d'exemple : les paragraphes sont indiqués par une seule lettre (`p`), de même que les liens (`a`), car, à l'origine, ces éléments étaient les plus fréquents dans les pages.

Nous devons ainsi trouver chaque fois un compromis entre lisibilité et concision. Si nous reprenons l'exemple de la mise à jour d'une liste déroulante du chapitre 4, où nous cherchions les villes de code postal 06850, nous pouvons produire le résultat suivant :

```
<?xml version="1.0" encoding="UTF-8"?>←❶
<communes cp="06850">
  <commune numero="06024" nom="Briançonnet"/>
  <commune numero="06063" nom="Gars"/>
  <commune numero="06116" nom="Saint-Auban"/>
</communes>
```

En ❶, nous précisons l'encodage, même si l'analyseur XML peut le deviner, car cette information est utile aux lecteurs humains. Nous indiquons dans la racine du document qu'il s'agit des communes de code postal 06850. Quant aux éléments `commune`, le nom de leurs attributs se passe de commentaires.

D'un autre côté, nous utilisons ce résultat d'abord pour fabriquer des listes déroulantes. Aussi avons-nous tout intérêt à lui donner une structure standard, quelles que soient les données qu'il représente, afin que notre composant `SelectUpdater` puisse l'interpréter de façon générique.

Ce résultat est en réalité simplement une liste d'éléments comprenant une valeur pour le navigateur et un texte pour l'utilisateur. Le nom de la liste importe peu, de même que les éventuels attributs qu'il pourrait avoir.

Nous choisissons donc plutôt de produire le résultat suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<communes cp="06850">
```

```
  <item value="06024" text="Briançonnet"/>
  <item value="06063" text="Gars"/>
  <item value="06116" text="Saint-Auban"/>
</communes>
```

Nous conservons l'élément racine, avec son attribut `cp`. Le contenu est ainsi explicite. Chaque élément de la liste porte le nom générique de `item` et a deux attributs : `value` et `text`, qui parlent d'eux-mêmes.

Modifions maintenant le fichier `get-villes-par-cp.php`, afin de produire ce résultat. Nous ajoutons un paramètre `output` à l'appel, de façon à pouvoir produire soit du texte simple, comme au chapitre précédent, soit du XML, soit enfin du JSON (que nous verrons ultérieurement).

Voici le code principal (le *main*, pourrait-on dire) :

```
sleep(1);
if (array_key_exists("cp", $_REQUEST)) {
  $tomorrow = 60*60*24; // en nb de secondes
  header("Cache-Control: max-age=$tomorrow");
  if (array_key_exists("output", $_REQUEST)) {
    if ($_REQUEST["output"] == "json") {
      print_json();
    }
    else if ($_REQUEST["output"] == "text") {
      print_plain();
    }
    else {
      print_usage();
    }
  }
  else {
    print_xml();
  }
}
else {
  print_usage();
}
```

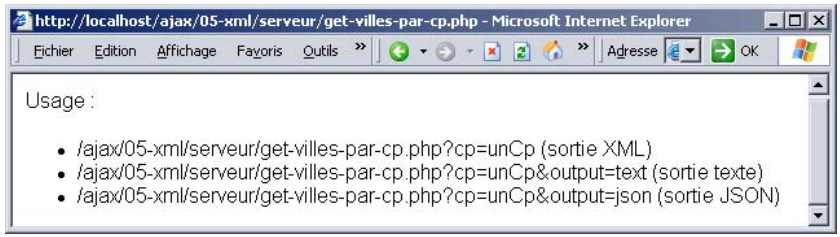
Nous avons simplement ajouté un branchement, selon que le paramètre `output` est présent ou non. S'il l'est, nous appelons `print_json` ou `print_text`, suivant la valeur de `output`. Sinon, nous appelons `print_xml`. XML est ainsi considéré comme la sortie par défaut.

Si `output` est incorrect ou si `cp` est absent, nous appelons la fonction `print_usage`, qui indique comment appeler la page :

```
function print_usage() {
  print "Usage : <ul>
    <li>$_SERVER[PHP_SELF]?cp=unCp (sortie XML)</li>
    <li>$_SERVER[PHP_SELF]?cp=unCp&output=text (sortie texte)</li>
    <li>$_SERVER[PHP_SELF]?cp=unCp&output=json (sortie JSON)</li>
  </ul>";
}
```

Le résultat est illustré à la figure 5.2.

Figure 5.2
Message lorsque la page est appelée sans les bons paramètres



La fonction `print_plain` demeure inchangée.

Voici la fonction `print_xml` :

```
function print_xml() {
  header("Content-Type: text/xml; charset=UTF-8");
  print "<?xml version='1.0' encoding='UTF-8'?>";❶
  print "<communes cp='$_REQUEST[cp]'>";
  $communes = get_communes($_REQUEST["cp"]);
  foreach ($communes as $commune) {
    $text = mb_convert_encoding($commune["text"], "UTF-8", "CP1252");❷
    print "<item value='$commune[value]' text=\"$text\"/>";❸
  }
  print "</communes>";
}
```

Les analyseurs XML sont plus stricts que les analyseurs HTML et ne comprennent pas l'encodage `windows-1252`. Aussi sommes-nous obligés d'utiliser `UTF-8`, d'où l'en-tête en ligne ❶, et le réencodage en ligne ❷. En ❸, nous mettons le texte entre doubles cotes et non entre apostrophes, car certains noms peuvent contenir une apostrophe. Le reste du code n'appelle pas de commentaire particulier.

Le résultat obtenu pour le code postal 06850 de notre exemple est illustré à la figure 5.3, où l'appel a été fait sans le paramètre `output`, comme l'indique la barre de titre de la fenêtre.

Figure 5.3
Résultat XML de la recherche de ville par code postal

Le DOM XML

En Ajax, lorsqu'une réponse HTTP est au format XML, l'arbre DOM résultant est disponible dans la propriété `responseXML` de l'objet `XMLHttpRequest`. Nous pouvons alors utiliser le DOM pour le manipuler.

La plus grande partie du DOM XML est identique au DOM HTML. Il y manque toutefois l'attribut si pratique `innerHTML`, qui n'a pas d'équivalent en XML, si bien que cette API est très lourde.

Les mécanismes permettant de passer de l'arbre à la représentation textuelle du document XML (sérialisation) et de la représentation à l'arbre (analyse) n'ont été normalisés qu'en avril 2004 et ne sont pas un modèle de simplicité. Ils ne sont pas non plus implémentés de façon portable. En Ajax, ce n'est heureusement pas gênant pour communiquer avec le serveur, car l'analyse est faite de façon transparente à travers l'attribut `responseXML` de `XMLHttpRequest`, et la sérialisation à travers sa méthode `send`, qui peut prendre en paramètre un arbre DOM.

Les principales classes du DOM XML sont illustrées à la figure 5.4.

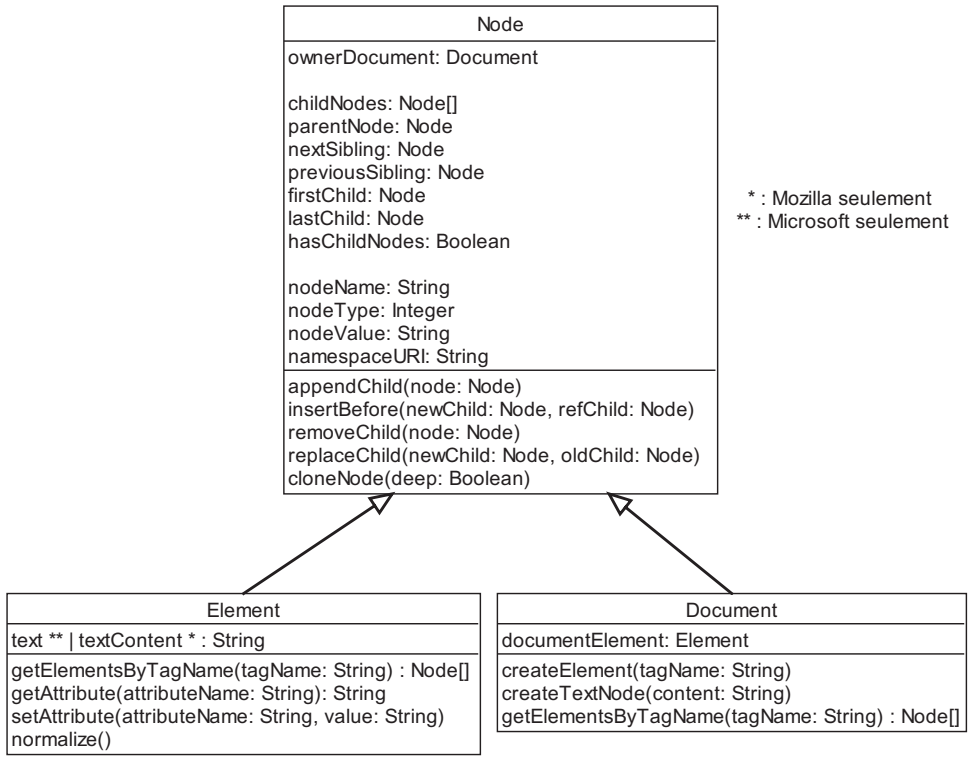


Figure 5.4

Principales classes du DOM XML

En comparaison du HTML, nous constatons que les attributs d'affichage (`offsetLeft`, etc.) ont disparu, ce qui est normal, puisque XML se cantonne à la structure.

`Node` possède deux attributs supplémentaires : `namespaceURI`, en liaison avec les espaces de noms, que nous aborderons avec XSLT, et `ownerDocument`. Dans une page HTML ou une application en général, il peut y avoir à un moment donné plusieurs documents XML en mémoire. Cet attribut permet de savoir à quel document un nœud est rattaché. Les autres attributs et méthodes de `Node` mentionnés sur le schéma sont identiques à leur contrepartie DOM HTML.

`Element` dispose d'un attribut `text` (dans IE) ou `textContent` (dans Mozilla), ce qui montre qu'il n'est guère portable. La bibliothèque JavaScript dojo en fournit toutefois une version portable sur tous les navigateurs, à travers sa fonction `dojo.dom.textContent`, qui calcule cet attribut et permet de le mettre à jour.

`Element` dispose en outre de la méthode `normalize`, qui devrait être très pratique, puisqu'elle est censée éliminer tous les nœuds texte vides et fusionner tous les nœuds texte adjacents descendant de l'élément courant. Cela permettrait de ne pas se soucier des nœuds texte vides, qui peuvent si facilement invalider les traitements qui s'appuient sur la structure du document. Ainsi, si nous reprenons l'exemple des communes précédent, nous ne pouvons supposer que `communes` a pour seuls enfants des éléments `commune` (il peut y avoir des enfants nœuds texte vide). Malheureusement, dans Mozilla et Opera, `normalize` n'élimine pas les nœuds vides, alors même que c'est son principal intérêt, L'impossibilité de l'utiliser occasionne beaucoup de lourdeur.

Par exemple, pour changer le contenu d'un élément, il faut supprimer tous ses enfants et écrire ensuite :

```
node = request.responseXML.createTextNode("blablabla");
element.appendChild(node);
```

La situation est en réalité beaucoup plus ennuyeuse encore, car si IE élimine par défaut les nœuds texte vides, dès la construction de `responseXML`, les autres navigateurs les conservent. Il est donc dangereux et non portable de se fonder sur l'ordre des enfants ou sur la valeur du contenu. C'est pourquoi nous recommandons d'utiliser des attributs plutôt que des éléments, l'accès aux attributs étant plus direct et se révélant plus sûr.

La mise à jour partielle avec XML

Nous allons transformer notre composant JavaScript `SelectUpdater` *(voir le chapitre 4)* pour lui faire prendre en compte les réponses XML ou JSON.

Commençons par modifier la méthode `onload`, en la rendant générique, et par créer trois méthodes : `loadText`, `loadXML` et `loadJSON`, qui en fournissent l'implémentation selon que la réponse est de type textuel, XML ou JSON.

Voici le code modifié de `onload` :

```
onload: function() {
  var hasContent, loadContent;←❶
```

```
  var type = ←❷
    this.request.getResponseHeader("Content-Type").split(";")[0];
  switch (type) {
    case "text/plain":case "text/html":←❸
      hasContent = (this.request.responseText != "");
      loadContent = this.loadText;
      break;
    case "text/javascript":case "application/json":←❹
      var items = this.request.responseText.parseJSON();
      hasContent = (items && items.length != 0);
      loadContent = this.loadJSON;
      break;
    case "text/xml":case "application/xml":←❺
      var root = this.request.responseXML.documentElement;
      hasContent = (root.childNodes.length > 0);
      loadContent = this.loadXML;
      break;
    defaut:←❻
      Log.error(this.url + " has an invalid Content-Type." +
        "Found '" + type + "' (must be plain text, JSON or XML)");
  }
  // Traitement commun
  this.select.innerHTML = "";←❼
  this.hide();
  if (hasContent) {
    loadContent.apply(this);←❽
    this.select.disabled = false;
  }
  else {
    this.msg.innerHTML = "<span style='color: red'>"
      + this.NOT_FOUND_MSG + "</span>";
    this.select.disabled = true;
  }
},
```

Cette méthode commence par positionner deux variables : un booléen (`hasContent`) et une fonction (`loadContent`), qui sont déclarés ligne ❶. Elle les utilise ensuite pour exécuter le même traitement, quel que soit le format de la réponse HTTP.

En ❷, elle récupère le type MIME de la réponse. Il est fondamental que le traitement serveur positionne celui-ci ; c'est pourquoi nous produisons en ❻ un message d'erreur si ce n'est pas le cas.

En ❸, nous retrouvons le cas traité auparavant, lorsque le type MIME indique du texte simple ou du HTML. Le booléen est très simple à déterminer : il y a du contenu quand la réponse textuelle `responseText` n'est pas vide.

Nous examinerons le cas JSON (débutant en ligne ❹) ultérieurement.

Nous détectons que la réponse est de type XML par les types MIME `text/xml` et `application/xml` (ligne ❻), et nous récupérons l'élément racine de la réponse dans la variable `root`.

Les booléens et la fonction de traitement étant positionnés, nous commençons en ❼ le traitement commun à tous les formats, lequel reprend celui effectué dans la première version de `SelectUpdater`. Notons simplement en ❽ l'appel à la méthode `apply` des fonctions, appel nécessaire pour que `this` soit positionné à l'objet courant dans l'appel `loadContent`.

La méthode `loadText` est maintenant réduite à :

```
loadText: function() {
  var options = this.request.responseText.split(";");
  var item, option;
  for (var i=0 ; i<options.length ; i++) {
    item = options[i].split("="); // value = text
    option = document.createElement("option");
    option.setAttribute("value", item[0]);
    option.innerHTML = item[1];
    this.select.appendChild(option);
  }
},
```

Terminons avec la méthode `loadXML` :

```
loadXML: function() {
  var root = this.request.responseXML.documentElement;
  var items = root.childNodes;
  var option;
  for (var i=0 ; i < items.length ; i++) {
    if (items[i].nodeName == "item") {❶
      option = document.createElement("option");
      option.setAttribute("value", items[i].getAttribute("value"));
      option.innerHTML = items[i].getAttribute("text");
      this.select.appendChild(option);
    }
  }
},
```

Nous récupérons l'élément racine de la réponse, puis nous parcourons ses enfants. Chaque fois que l'un d'eux est de type `item` (ligne ❶), nous produisons un élément `option` afin d'éliminer les nœuds texte vides éventuels.

JSON (JavaScript Object Notation)

Comme expliqué au chapitre 3, JSON est la notation objet de JavaScript, qui permet de représenter sous forme textuelle toute variable JavaScript.

Cette forme s'appuie sur les deux structures suivantes :

- une collection de couples nom/valeur, représentant un objet JavaScript ou, dans d'autres langages, un enregistrement, une structure, un dictionnaire, une table de hachage ou un tableau associatif ;
- une liste de valeurs ordonnées, représentant un tableau JavaScript ou, dans d'autres langages, un vecteur ou une liste.

Ces structures de données sont universelles. Elles se rencontrent, sous une forme ou une autre, dans tous les langages de programmation modernes (Java, Eiffel, C++, PHP, Delphi, Ruby, Perl, Python, etc.). JSON constitue ainsi un format léger d'échange de données entre applications, qui se révèle dans ce domaine un concurrent de XML, et ce, quel que soit le langage de programmation dans lequel ces applications sont écrites.

Il suffit de disposer dans chaque langage d'une API pour encoder et décoder du JSON. Le site *www.json.org* présente en détail cette notation et recense les API disponibles dans les différents langages.

Dans le cas de langages à base de classes, notons que s'il est tout aussi facile et direct de transformer des objets en notation JSON, il faut plus de travail pour convertir une chaîne JSON en un objet d'un type classe. La chaîne JSON ne transporte en effet aucune information sur le typage de son contenu : les objets ne se réduisent pas à leurs données (à leurs champs), et ils ont aussi un type, avec tout ce que cela induit en méthodes, visibilité (private/public) ou encore héritage. En résumé, JSON transporte des données, et non des objets.

Le langage se résume aux formes suivantes :

- Un objet : ensemble de couples nom/valeur non ordonnés. Un objet commence par `{` (accolade gauche) et se termine par `}` (accolade droite). Chaque nom est suivi de `:` (deux-points) et les couples nom/valeur sont séparés par `,` (virgule).
- Un tableau : collection de valeurs ordonnées. Un tableau commence par `[` (crochet gauche) et se termine par `]` (crochet droit). Les valeurs sont séparées par `,` (virgule).
- Une valeur : soit une chaîne de caractères entre guillemets, soit un nombre, soit `true`, `false` ou `null`, soit un objet, soit un tableau. Ces structures peuvent être imbriquées.
- Une chaîne de caractères : suite de zéro ou *n* caractères Unicode, entre guillemets, et utilisant les échappements avec antislash. Un caractère est représenté par une chaîne d'un seul caractère.

Il est possible d'inclure des espaces entre les éléments pour clarifier la structure.

L'exemple des communes, dont la forme XML est la suivante :

```
<communes cp='06850'>
  <item value='06024' text='Briançonnet'/>
  <item value='06063' text='Gars'/>
  <item value='06116' text='Saint-Auban'/>
</communes>
```

pourrait prendre en JSON la forme suivante :

```
{"communes": {
  "cp": "06850",
  "items": [←❶
    { "value":"06024", "text":"Brian\u00e7onnet" },←❷
    { "value":"06063", "text":"Gars" },
    { "value":"06116", "text":"Saint-Auban" }
  ]
}
```

Nous constatons combien les deux formes sont proches. La forme JSON, semblable à celle des programmes dans les langages dérivés du C, semble familière aux développeurs et est donc facilement lisible par eux. Certains la trouvent plus explicite que la forme XML, tandis que, pour d'autres, c'est le contraire. Certains prétendent que XML est plus verbeux que JSON. Lorsque tout est mis en éléments, c'est vrai. Mais lorsque les attributs sont privilégiés, comme ici, XML semble tout aussi concis.

Notons cependant quelques différences : les éléments `item` du code XML ont pour contrepartie JSON un champ `items` unique, dont la valeur est un tableau d'objets (ligne ❶). C'est obligatoire, car il est impossible de donner le même nom d'attribut à deux champs d'un objet.

Par ailleurs, le caractère « ç » de Briançonnet a été remplacé en JSON par sa représentation Unicode `\u00e7` (ligne ❷), qui garantit sa portabilité.

Communication JSON entre JavaScript et PHP

En PHP, il est très facile d'encoder/décoder en JSON des tableaux et tableaux associatifs grâce à la classe `Services_JSON` définie dans le fichier **JSON.php,** disponible à l'adresse *http://pear.php.net/pepr/pepr-proposal-show.php?id=198*.

Voici comment utiliser cette classe pour encoder en JSON :

```
// Une variable PHP complexe
$bach = array(
  "prenom" => "Johann Sebastian",
  "nom" => "Bach",
  "enfants" => array(
    "Carl Philipp Emanuel",
    "Johan Christian")
);
// Instancier la classe Services_JSON
$json = new Services_JSON();
// Convertir la variable complexe
print $json->encode($bach);
```

Ce qui produit :

```
{"prenom":"Johann Sebastian","nom":"Bach","enfants":["Carl Philipp Emanuel","Johan
Christian"]}
```

Il suffit d'une ligne pour instancier le service d'encodage et d'une autre pour encoder une variable. Dans l'autre sens, c'est tout aussi facile.

Prenons la valeur obtenue par l'encodage, et décodons-la :

```
$bach='{"prenom":"Johann Sebastian","nom":"Bach","enfants":["Carl Philipp Emanuel","Johan
Christian"]}';
print_r($json->decode($bach));
```

Cela produit :

```
(
  [prenom] => Johann Sebastian
  [nom] => Bach
  [enfants] => Array
    (
      [0] => Carl Philipp Emanuel
      [1] => Johan Christian
    )
)
```

Là encore, une instruction suffit à décoder. En PHP, c'est donc un jeu d'enfant que de transformer des objets et tableaux en JSON, et réciproquement.

En JavaScript, ces transformations sont encore plus simples, grâce au fichier **json.js,** disponible à l'adresse *http://www.json.org/json.js,* qui les ajoute au langage :

- Il ajoute au prototype de `Array` et de `Object` la méthode `toJSONString`, qui produit une représentation JSON du tableau courant ou de l'objet courant.
- Il ajoute au prototype de String la méthode `parseJSON`, qui analyse une chaîne JSON et renvoie l'objet JavaScript correspondant ou `false` si la chaîne est incorrecte.

Ainsi, si nous écrivons :

```
bach = {
  "prenom": "Johann Sebastian",
  "nom": "Bach",
  "enfants": [
    "Carl Philip Emmanuel",
    "Wilhelm Gottfried"]
}
log(bach.toJSONString());
```

nous obtenons :

```
{"prenom":"Johann Sebastian","nom":"Bach","enfants":["Carl Philip Emmanuel","Wilhelm
Gottfried"]}
```

Inversement, pour récupérer en JavaScript une réponse JSON, nous écrivons :

```
var unObjet = request.responseText.parseJSON();
```

Nous pourrions penser que cette méthode `parseJSON` est inutile, puisque l'instruction :

```
var unObjet = eval(request.responseText);
```

semble avoir le même effet.

C'est certes le cas quand la réponse est du JSON correct, mais elle pourrait être incorrecte, auquel cas une exception serait levée, ou, pire encore, elle pourrait contenir du code exécutable, qui serait alors exécuté. Cela pourrait être dangereux.

Au contraire, la méthode `parseJSON` évalue l'expression seulement si c'est une chaîne JSON, renvoyant `false` dans le cas contraire, ou si elle est mal formée. C'est ainsi une précaution importante pour la sécurité de nos applications.

Nous pouvons maintenant aller plus loin encore dans l'intégration JavaScript/PHP en transmettant au serveur des objets JavaScript dans le corps des requêtes `XMLHttpRequest`.

Nous écrivons côté client :

```
// Un objet JavaScript
bach = {
  "prenom": "Johann Sebastian",
  "nom": "Bach",
  "enfants": [
    "Carl Philipp Emanuel",
    "Johann Christian"]
}
request = new XMLHttpRequest();
request.open("POST", "serveur/lire-json.php", false);
request.setRequestHeader("Content-type",
  "application/x-www-form-urlencoded");
// Envoyer l'objet en JSON
request.send(bach.toJSONString());←❶
log(request.responseText);
```

Nous créons un objet JavaScript et envoyons sa représentation JSON en tant que corps de la requête. L'action côté serveur (`lire-json.php`) se contente de décoder le corps et l'affiche sous forme détaillée.

Voici son code :

```
// Recuperer les donnees POST, censees etre en JSON
$input = file_get_contents('php://input');←❷
$value = $json->decode($input);
print_r($value);
```

La chaîne `php://input` désigne le corps de la requête HTTP. La ligne ❷ place ainsi dans `$input` la chaîne JSON transmise en ❶, et il suffit dès lors de la décoder.

L'encodage/décodage JSON est rapide, notamment côté client, et donne accès directement aux variables. La communication en JSON peut être ainsi une solution avantageuse, ce qui explique qu'elle gagne de plus en plus de faveur.

La mise à jour partielle avec JSON

Nous allons porter notre mise à jour en JSON. Le traitement principal appelle la fonction `print_json` lorsque le paramètre `output` vaut `json`.

Voici le code de cette fonction :

```
function print_json() {
  include_once("JSON.php");
  $json = new Services_JSON();←❶
  $communes = get_communes($_REQUEST["cp"]);←❷
  // On ne peut pas utiliser foreach qui travaille par copie
  // et non par reference
  for($i=0 ; $i<count($communes) ; $i++) {
    $communes[$i]["text"] =←❸
      mb_convert_encoding($communes[$i]["text"],"UTF-8", "CP1252");
  }←❹
  $result = array("cp" => $_REQUEST["cp"], "items" => $communes);
  header("Content-type: text/javascript");
  print $json->encode($result);←❺
}
```

Nous commençons par instancier en ❶ le service d'encodage, après avoir inclus le fichier qui le contient. Nous récupérons en ❷ le résultat, qui est un tableau de paires `value`/`text`.

Si nous n'avions pas de problème de jeux de caractères, nous irions directement en ❹, et le code serait vraiment très simple. Malheureusement, nous devons nous en préoccuper, ce que nous faisons en ❸. Nous produisons un tableau associatif comportant deux clés : `cp` et `items`. Il ne reste plus qu'à l'encoder en JSON (ligne ❺), après avoir indiqué l'entête de type MIME adéquat.

Voici le résultat produit :

```
{"cp":"06850","items":[{"value":"06024","text":"Brian\u00e7onnet"},{"value":"06063","text":
"Gars"},{"value":"06116","text":"Saint-Auban"}]}
```

ce qui, mis en forme, donne :

```
{
  "cp":"06850",
  "items":[
    {"value":"06024", "text":"Brian\u00e7onnet"},
    {"value":"06063","text":"Gars"},
    {"value":"06116","text":"Saint-Auban"}
  ]
}
```

Le résultat est un peu moins explicite qu'avec XML, la balise `<communes cp="06850">` n'ayant ici que l'attribut `cp` pour contrepartie. Ce n'est pas inhérent à JSON, et nous aurions pu ajouter un attribut `data` valant `communes`. Comme en XML, ces attributs sont libres, le seul qui est imposé par notre composant `SelectUpdater` étant l'attribut `items`.

Dans le composant `SelectUpdater`, revenons sur la méthode `onload` :

```
onload: function() {
  var hasContent, loadContent;
  var type =
```

```
    this.request.getResponseHeader("Content-Type").split(";")[0];
  switch (type) {
    case "text/plain":case "text/html":
    // (code sans interet ici)
    case "text/javascript":case "application/json":
      var result = this.request.responseText.parseJSON();←❶
      try {
        hasContent = (result.items.length != 0);←❷
      }
      catch (exc) {
        hasContent = false;
        Log.error(this.url + " ne renvoie pas un tableau JSON");←❸
      }
      loadContent = this.loadJSON;
      break;
     //etc.
```

Nous récupérons en ❶ l'objet correspondant à la réponse JSON. Lorsque la réponse est du JSON correct, l'attribut `items` de la variable `result` vaut un tableau. Il suffit de regarder sa taille pour savoir si la requête donne un vrai résultat (ligne ❷). Dans le cas où la réponse n'est pas du JSON correct, une exception est levée, et nous produisons un message d'erreur (ligne ❸).

Il ne reste plus qu'à examiner `loadJSON` :

```
loadJSON: function() {
  var items = this.request.responseText.parseJSON().items;←❶
  var item, option;
  for (var i=0 ; i<items.length ; i++) {
    option = document.createElement("option");
    option.setAttribute("value", items[i].value);
    option.innerHTML = items[i].text;
    this.select.appendChild(option);
  }
},
```

Nous récupérons les items en ❶, puis nous créons un élément `option` pour chacun d'eux, comme nous l'avons fait pour les méthodes `loadXML` et `loadText`.

Comparaison des formats d'échange

Avec la mise à jour de la liste déroulante, nous avons utilisé successivement trois types de résultats : textuel, XML et JSON. La question du meilleur choix se pose donc tout naturellement.

Dans deux cas de figure, la réponse est simple :

- Lorsque les données consistent uniquement en une liste de valeurs, il est inutile de les structurer, et le plus simple est sans doute de les envoyer comme une simple chaîne de caractères, chaque valeur étant séparée des suivantes par un caractère spécial, la fin de

ligne de préférence, car c'est le terminateur naturel. Nous pourrions certes les produire en XML ou en JSON, mais cela demanderait un travail de mise en forme côté serveur et d'analyse côté client. Il semble donc judicieux de s'en dispenser et de privilégier la sortie textuelle. C'est précisément ce que nous avons fait dans notre suggestion de saisie.

- Lorsque les données consistent en un document, comme un flux RSS, là encore la réponse est simple : XML s'impose. Il est conçu pour cela et y est donc bien adapté. Au contraire, une sortie textuelle perdrait toute la structure. Quant à JSON, il ne conserverait pas la distinction attribut/élément qui a une signification précise en XML, si bien que nous perdrions une partie de l'information.

Un troisième cas de figure concerne les données structurées. Dans ce cas, la sortie textuelle est sans intérêt, puisqu'elle élimine la structure. Par contre, tant JSON que XML peuvent convenir et sont donc concurrents. C'est normal : JSON est conçu pour cela, et XML en partie aussi (en partie, car il a été conçu aussi pour le domaine des documents).

Notre choix tient en ce cas à plusieurs facteurs : la simplicité du code induit, son efficacité côté client comme côté serveur, la portabilité, dans le cas où nous voudrions faire de notre action côté serveur un service Web, et la facilité d'exploitation des données.

Du point de vue de la simplicité du code, JSON semble avoir l'avantage côté serveur, si les données dont nous partons ont la forme que nous voulons transmettre. C'était le cas avec la liste des communes. Côté client, JSON et XML sont équivalents, à condition que le résultat XML place les données dans des attributs plutôt que dans le contenu. Nous le constatons avec l'exemple des communes.

Du point de vue de l'efficacité, on considère que JavaScript analyse beaucoup plus vite du JSON que du XML, certains avançant un facteur de 10. Toutefois, il n'est pas évident que cela fasse une différence significative, car avec JSON comme avec XML, il faut de toute façon faire des manipulations DOM de l'arbre HTML, et c'est cela qui est le plus lent, puisque le navigateur doit non seulement mettre à jour l'arbre, mais en plus recalculer son affichage. En revanche, côté serveur, JSON peut être plus efficace, évitant d'y maintenir un arbre DOM.

Du point de vue de la portabilité, les deux formats sont techniquement utilisables avec la plupart des langages, et les résultats peuvent être édités dans la plupart des éditeurs de texte. C'est en fait surtout une question d'habitude. XML est actuellement beaucoup plus connu que JSON, et c'est le format standard sous-tendu par les services Web.

C'est l'exploitation des données qui différencie surtout les deux formats. Avec XML, il est possible de produire assez simplement une représentation HTML, voire PDF, des données, grâce à XSLT, tandis qu'avec JSON, on est obligé de faire du DOM, ce qui est très lourd. XML offre là un réel avantage. Reste à savoir dans quels cas nous avons besoin d'une telle représentation HTML.

Pour les données structurées, le choix entre JSON et XML est ainsi ouvert.

Il reste un dernier cas de figure : lorsque le serveur produit un fragment HTML, qui remplace purement et simplement le `innerHTML` d'un élément donné de la page. Nous avons utilisé cette technique au chapitre 1, lorsque le HTML retourné consistait en un message.

Ici, il ne s'agit ni de données, qui pourraient être utilisées dans un autre contexte, ni de documents autonomes, mais d'une partie de la page. L'intérêt est de produire un code JavaScript si simple et si parfaitement générique que nous puissions le remplacer par un appel à un composant JavaScript générique, que nous pourrions même étendre pour lui faire mettre à jour le fragment HTML à intervalles réguliers. La bibliothèque prototype propose précisément ces deux composants. C'est pratique et rapide à mettre en place.

En contrepartie, l'action côté serveur et la page sont fortement couplées. Il faut être vigilant sur ce couplage si nous voulons garantir un code simple et maintenable. Comme pour toute action côté serveur, il est préférable de séparer le traitement qui récupère, et met éventuellement à jour, les données (il s'agit de la partie modèle dans le MVC), de la partie qui met ce résultat en forme pour l'envoyer sur le client (partie vue du MVC). Si ce principe est respecté, nous restreignons le couplage de l'action avec la page à sa partie vue, ce qui est tout à fait acceptable.

Cette technique légère ne fonctionne cependant pas avec certains éléments HTML, `innerHTML` étant dans IE en lecture seule pour les éléments `html`, `select`, `table`, `thead`, `tbody`, `tfoot`, `tr` et `textarea`. C'est pourquoi nous ne l'avons pas utilisée pour la liste déroulante.

En conclusion, la réponse d'un appel `XMLHttpRequest` est loin d'être obligatoirement en XML, malgré son nom trompeur. En fait, selon le cas de figure, il peut être préférable de renvoyer du texte, un fragment HTML, un document XML ou du code JSON. Dans deux cas de figure, le choix est direct, tandis que, pour les autres, il faut regarder au cas par cas, en considérant les quelques critères bien définis que nous venons de détailler.

Exemple d'application avec les flux RSS

Après avoir étudié l'échange de données en XML et JSON, nous en venons à une application aussi emblématique du Web 2.0 que la suggestion de saisie : les flux RSS (RSS feed).

RSS, acronyme de Really Simple Syndication (syndication vraiment simple) ou de Rich Site Summary (sommaire d'un site enrichi), permet de diffuser des nouvelles sous une forme structurée, que l'on nomme flux RSS.

Des clients RSS, intégrés parfois dans le navigateur — c'est le cas d'Opera et de Firefox — permettent de les parcourir sous une forme conviviale. Le lecteur d'Opera, illustré à la figure 5.5, les présentent comme les fils de discussion des newsgroups, ce qui est pertinent, puisque les flux RSS comme les newsgroups diffusent des nouvelles par thème.

Un panneau affiche la liste des en-têtes, composés principalement de l'émetteur (dans l'exemple, le site *techno-sciences.net*), du sujet de la nouvelle et de sa date de publication. Un deuxième panneau affiche le résumé de l'article, ainsi qu'un lien vers l'article complet sur le site l'ayant émis.

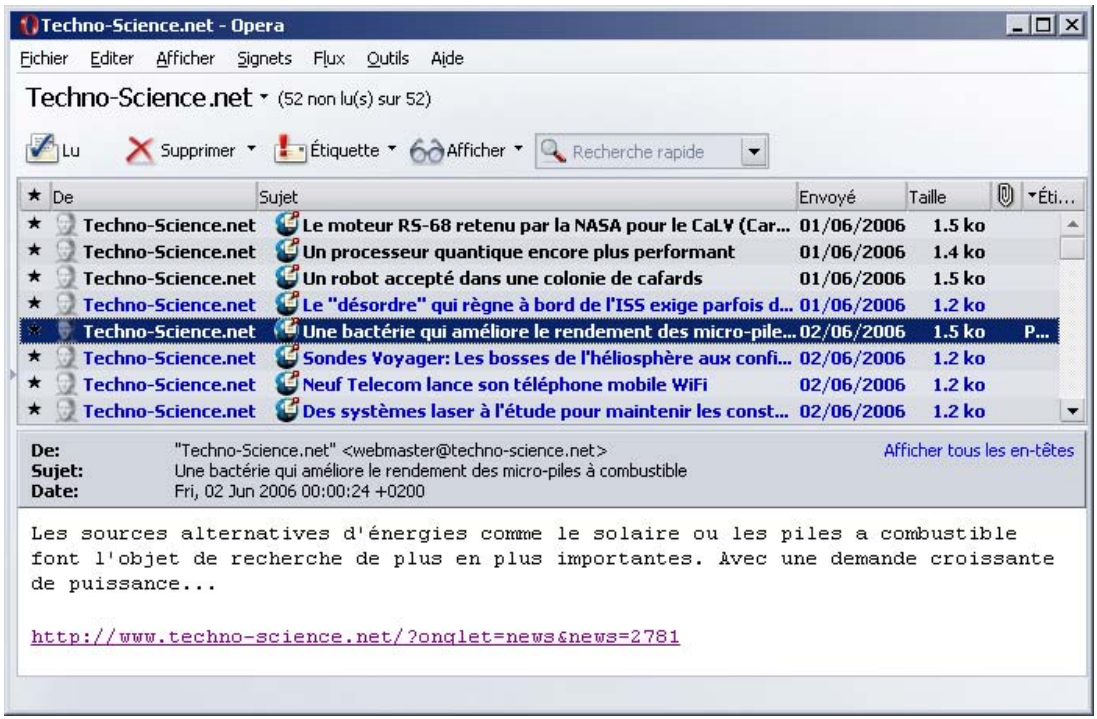


Figure 5.5 '

Le lecteur de RSS intégré dans Opera

Il devient fréquent dans le Web 2.0 d'inclure dans ses pages un ou plusieurs flux RSS. Par exemple, sur le site *netvibes.com,* l'utilisateur peut ajouter les flux qu'il désire en indiquant simplement leur URL. Chaque flux apparaît dans une boîte qu'il est possible de déplacer sur la page.

Nous allons réaliser un lecteur RSS simple, qui affichera dans un élément HTML un flux RSS. Nous pourrons placer celui-ci dans une boîte d'information, à la façon de *netvibes.com,* ou dans des onglets en utilisant les composants graphiques développés au chapitre 3.

Les formats RSS

Il existe sept formats différents de RSS, ceux-ci ayant évolué dans le temps, dont RSS 2.0 et Atom sont aujourd'hui les principaux.

Voici un exemple de flux RSS 2.0, provenant de *www.techno-sciences.net,* qui commence par l'en-tête :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<rss version="2.0">
```

```
  <channel>
    <title>Techno-Science.net</title>
    <link>http://www.techno-science.net</link>
    <description>Actualité des sciences et des techniques</description>
    <language>fr</language>
    <managingEditor>webmaster@techno-science.net</managingEditor>
    <webMaster>webmaster@techno-science.net</webMaster>
    <pubDate>Thu, 08 Jun 2006 00:01:33 +0200</pubDate>
    <image>
       <url>http://www.techno-science.net/graphisme/Logo/Logo-TSnet-noir-150x62.gif</
url>
      <title>Techno-Science.net</title>
      <link>http://www.techno-science.net</link>
      <width>150</width>
      <height>62</height>
    </image>
```

L'élément racine `rss` possède un attribut `version` utile au lecteur de RSS. Son premier enfant, `channel`, a quelques enfants donnant des informations sur le flux : nom et URL du site émetteur, dates et informations éditoriales. Les éléments `title`, `link` et `description` sont obligatoires, les autres optionnels.

Suivent une série d'articles, de nom de balise `item` :

```
    <item>
       <title>Une bactérie qui améliore le rendement des micro-piles à combustible</
title>
      <link>http://www.techno-science.net/?onglet=news&amp;news=2781</link>
      <description>Les sources alternatives d'énergies comme le solaire ou les piles
a combustible font l'objet de recherche de plus en plus importantes. Avec une demande
croissante de puissance...</description>
      <category>Energie</category>
      <pubDate>Fri, 02 Jun 2006 00:00:24 +0200</pubDate>
    </item>

    <!-- ici d'autres elements item -->
  </channel>
</rss>
```

Chaque article peut posséder, entre autres, un titre, un lien où le consulter en intégralité, un résumé, ainsi qu'une date de parution. Tous ces éléments sont optionnels, à l'exception de `title` et `description`, dont l'un au moins doit être présent.

La description complète des formats RSS est disponible à l'adresse *www.rssboard.org*.

Comme indiqué précédemment, toute l'information destinée à l'utilisateur réside dans le contenu des éléments, et non dans leurs attributs. Le seul attribut de tout le fichier, l'attribut `version` de l'élément `rss`, est utile aux applications lisant les RSS, mais non à l'utilisateur.

Il est possible d'éditer des RSS de façon visuelle (sans avoir à écrire les balises), grâce à l'extension Firefox RSS Editor.

Le serveur comme mandataire HTTP

Pour afficher en Ajax un flux RSS de notre choix, nous devons d'emblée faire face à une difficulté : par sécurité, les appels `XMLHttpRequest` ne peuvent être émis que vers le serveur ayant envoyé la page courante, alors que nous voulons récupérer un flux venant le plus souvent d'un autre serveur.

La solution à ce problème consiste à demander à notre serveur Web de servir de mandataire : nous lui demandons une page mandataire, chargée de récupérer l'URL passée en paramètre, et de nous la retourner.

Pour réaliser cela en PHP, nous utilisons la bibliothèque CURL (Client URL).

Voici le code principal de la page mandataire **open-url.php :**

```
if (array_key_exists("url", $_REQUEST)) {
  afficher($_REQUEST["url"]);
}
else {
  demander();
}
```

Si la page reçoit un paramètre `url`, elle appelle la fonction `afficher`, qui récupère et renvoie la page située à cette URL. Sans paramètre, elle appelle la fonction `demander`, qui affiche un formulaire pour saisir cette URL.

Comme nous ne détaillerons pas la fonction `demander`, sans importance ici, passons directement à la fonction `afficher`, dont voici le code :

```
function afficher($url) {
  // Initialiser la recuperation
  $ch = curl_init($url);
  // Incorporer les en-tetes (et le statut)
  curl_setopt($ch, CURLOPT_HEADER, 1);
  // Renvoyer le resultat dans une chaine
  // et non sur la sortie standard
  curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
  // Executer la requete
  $page = curl_exec($ch);←❶
  curl_close($ch);
  // Les en-tetes sont terminees par 2 CRLF
  $endHeader = strpos($page, "\r\n\r\n");
  // Recuperer en-tetes et corps de la reponse
  $headers = substr($page, 0, $endHeader);←❷
  $body = substr($page, $endHeader+4);
  // Chaque en-tete est terminee par CRLF
  $headers = explode("\r\n", $headers);←❸
  foreach ($headers as $line) {
    header("$line\n");←❹
  }
  // Produire le corps
  print $body;←❺
}
```

En ❶, nous récupérons dans la variable `$page` la réponse complète, avec son corps et ses en-têtes, car nous voulons renvoyer exactement ce qu'aurait renvoyé au client un appel direct à l'URL demandée.

Il s'agit ensuite d'envoyer séparément les en-têtes et le corps. En effet, en PHP, les premières sont envoyées par la fonction `header`, tandis que le corps l'est par `print` (ou `echo`, qui est un synonyme). Chaque en-tête étant terminé par `CRLF` (retour chariot puis fin de ligne), et le dernier étant suivi d'une ligne vide, nous extrayons de `$page` les en-têtes `$headers` (ligne ❷) et le corps `$body` (ligne suivante). Puis, nous faisons un tableau des en-têtes et envoyons chacun au client (ligne ❹). Il ne reste plus qu'à envoyer le corps (ligne ❺).

Simple à utiliser, la bibliothèque CURL se révèle très pratique, non seulement pour notre besoin présent, mais aussi pour appeler des actions distantes et intégrer des données XML distribuées.

Le composant lecteur de RSS

Nous souhaitons faire afficher un flux RSS dans un élément HTML. Les deux paramètres importants sont cet élément (ou son id) et l'URL du flux RSS. Le reste étant parfaitement générique, nous allons créer un composant JavaScript, que nous nommerons `RSSBox`.

L'appel dans la page sera de la forme suivante :

```
uneBoiteRSS = new RSSBox(urlDuFlux, idElementHTML);
```

Voici le constructeur de `RSSBox` :

```
function RSSBox(url, idElement, maxNumber, target) {
  if (!window.XMLHttpRequest) {←❶
    throw "Requiert XMLHttpRequest()";
  }
  /** URL du flux RSS */
  this.url = url;
  /** Elément affichant le résultat HTML */
  this.element = document.getElementById(idElement);
  /** Requête pour le flux RSS */
  this.request = null;
  /** Nombre maximal de nouvelles affichées (défaut 10)*/
  this.maxNumber = maxNumber || 10;
  /** Nom de la fenêtre où renvoyer les liens (défaut _self)*/
  this.target = target || "_self";
  this.load();←❷
}
```

Les commentaires indiquent la signification des différents attributs. En ❶, nous vérifions qu'Ajax est disponible, lançant une exception dans le cas contraire. Une fois l'initialisation terminée, nous lançons en ❷ la récupération du flux.

Celle-ci est définie dans le prototype, qui définit en outre un attribut `proxyUrl` et des méthodes `reload`, `display` et `_setTitle`, cette dernière étant privée, comme l'indique le

souligné au début de son nom. L'attribut `proxyUrl` spécifie l'URL de la page sur le serveur récupérant les flux distants.

Voici le code du prototype, épuré des commentaires jsdoc :

```
RSSBox.prototype = {
  proxyUrl: "serveur/open-url.php?url=",
  load: function(time) { // voir ci-dessous },
  reload: function() { // voir ci-dessous },
  display: function() { // voir ci-dessous },
  _setTitle: function() { // voir ci-dessous }
}

RSSBox.prototype.constructor = RSSBox;
```

L'attribut `proxyUrl` vaut par défaut l'URL relative de la page que nous avons définie précédemment, avec le nom de son paramètre `url`. Cet attribut est défini dans le prototype et non dans les instances du constructeur, car toutes les instances doivent partager cette valeur, qui est constante pour une application.

La méthode `load`, chargée de lancer la requête, prend un paramètre optionnel `time`, dont nous comprendrons l'utilité en regardant la méthode `reload`.

Voici son code :

```
load: function(time) {
  // Annuler la requete si elle est en cours
  if (this.request) {
    try {
      this.request.abort();←❶
    }
    catch (exc) {}
  }
  this.request = new XMLHttpRequest();
  var rssUrl = this.proxyUrl + encodeURIComponent(this.url); ←❷
  if (time) {
    rssUrl += "&time="+now;←❸
  }
  this.request.open("GET", rssUrl, true);
  var current = this;
  this.request.onreadystatechange = function() {
    if (current.request.readyState == 4) {
      if (current.request.status == 200) {
        current.display();←❹
      }
      else {
        current.element.innerHTML =
          "Impossible de récupérer <a href='"
          + current.url + "'>ce flux RSS</a> (http status : "
          + current.request.status + ")";←❺
      }
    }
  }
```

```
    this.element.innerHTML = "En chargement ...";←❻
    this.request.send("");
  },
```

En ❶, nous annulons la requête en cours, s'il y en a une, comme nous l'avons fait jusqu'ici. Dans ce cas, nous aurions pu nous en dispenser sans grand mal, car le risque d'avoir deux requêtes parallèles pour le même objet `RSSBox` est faible.

En ❷, nous encodons l'URL du flux RSS à récupérer. C'est indispensable, car cette URL contient, en tant qu'URL, des caractères interdits dans les paramètres.

En ❸, si la méthode a été appelée avec un paramètre, nous utilisons le mécanisme décrit au chapitre précédent : nous ajoutons à l'URL demandée un paramètre dont la valeur vaut l'instant courant, de sorte que l'URL obtenue ne peut avoir déjà été demandée et par conséquent ne se trouve pas dans le cache. Nous récupérons ainsi la dernière version du flux, que celui-ci ait ou non été mis en cache précédemment.

À l'envoi de la requête, ou plutôt juste avant, nous informons l'utilisateur (ligne ❻). Lors de la réception de la réponse, nous affichons le résultat si tout s'est bien passé (ligne ❹) ou un message d'erreur indiquant la nature du problème s'il y en a eu un (ligne ❺).

Voici le code de la méthode `reload` :

```
reload: function() {
  var now = (new Date()).getTime();
  this.load(now);
},
```

Cette méthode recharge le flux, en outrepassant sa mise en cache éventuelle. Elle est utile dans IE et Opera, où elle fournit la fonctionnalité Recharger la page, classique en Web traditionnel, dont la contrepartie Ajax (Recharger le fragment de page) n'est pas fournie par le navigateur et doit être écrite par le développeur.

Voici la méthode `display` :

```
display: function() {
  this._setTitle();←❶
  this.element.innerHTML = "";←❷
  var ele = this.request.responseXML.documentElement;
  ele = Element.getChildElements(ele, "channel")[0];←❸
  var items = Element.getChildElements(ele, "item");
  var i, link, length;  var i, ele, link, length;
  length = Math.min(items.length, this.maxNumber);
  for (i=0 ; i < length ; i++) {
    // Creer un lien par item
    link = document.createElement("a");←❹
    // Avec l'url trouvee dans link
    ele = Element.getChildElements(items[i], "link")[0];←❺
    link.setAttribute("href", Element.textContent(ele));←❻
    link.setAttribute("target", this.target);
    // Le texte venant de title
    ele = Element.getChildElements(items[i], "title")[0];
```

```
    link.innerHTML = Element.textContent(ele);
    // Le lien en affichage block
    link.style.display = "block";
    this.element.appendChild(link);
  }
},
```

Cette méthode commence par récupérer le titre du flux (ligne ❶) et par effacer le contenu de l'élément affichant le flux (ligne ❷). Pour récupérer les articles, il faut partir de l'élément racine, descendre sur le premier élément `channel` (ligne ❸) et en prendre les enfants de type `item`.

Nous aurions pu écrire :

```
var items = this.request.responseXML.getElementsByTagName("item");
```

mais nous prenons toutes les précautions possibles, dans la mesure où certains flux ne sont pas complètement conformes au standard. Le flux de techno-science.net lui-même ne passe pas avec succès la validation en ligne proposée par le site *www.rssboard.org,* mentionné précédemment (il viole d'autres règles que celles concernant les items).

Nous utilisons en ❸ et en ❺ la méthode `getChildElements` définie dans notre fichier **util.js** déjà mentionné à maintes reprises. Nous créons un lien hypertexte par article (ligne ❹) puis utilisons en ❻ la méthode `textContent`, définie elle aussi dans **util.js** et qui provient de la bibliothèque dojo.

Il a longtemps été impossible de récupérer le contenu textuel d'un élément en DOM XML. C'est pourquoi Microsoft avait ajouté l'attribut `text`. Quand le W3C s'est enfin avisé qu'une telle propriété pouvait être utile, il l'a ajouté, mais avec un nom différent : `textContent`. Malheureusement, tous les navigateurs ne comprennent pas encore cette propriété, si bien que dojo l'a implémenté par une méthode qui construit le résultat en balayant les descendants de l'élément courant.

Les figure 5.6 et 5.7 illustrent la documentation jsdoc de `RSSBox`. Nous remarquons que l'outil rend dans le paragraphe Requires les dépendances indiquées dans les commentaires.

Affichage du flux

Il nous reste à utiliser ce composant dans une page. Nous allons placer le flux affiché dans une boîte d'information, telle que nous l'avons vue au chapitre 3.

Voici le code HTML de la page :

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="RSSBox.css"/>
    <script type="text/javascript" src="util.js"></script>←❶
    <script type="text/javascript" src="RSSBox.js"></script>
    <script type="text/javascript" src="InfoBox.js"></script>
    <title>Lire des flux RSS</title>
```

Figure 5.6
Attributs du composant RSSBox

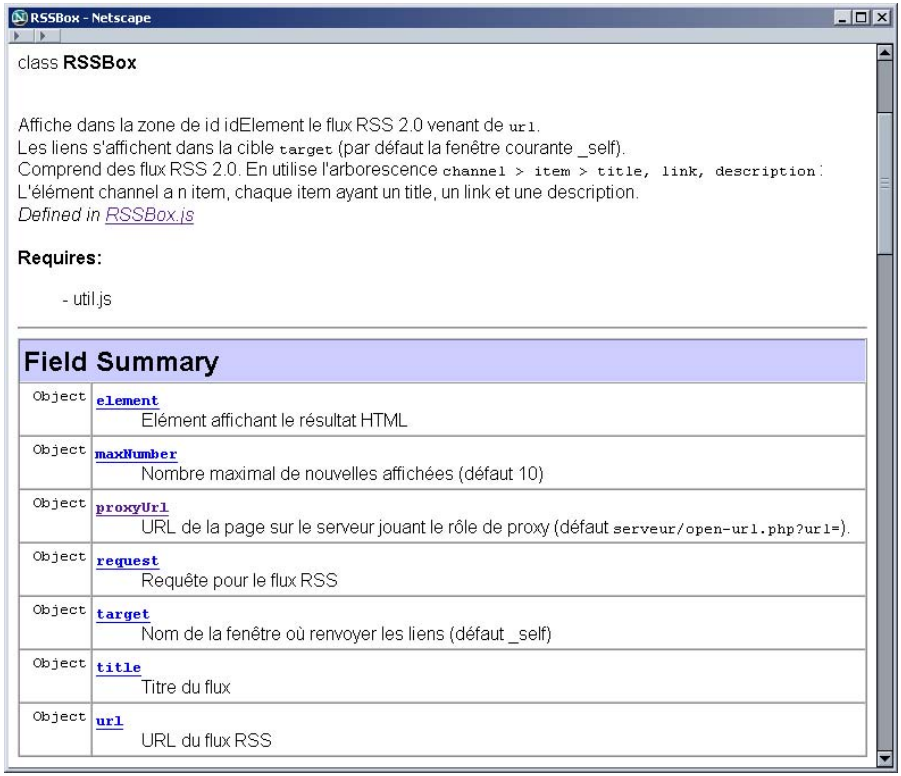


Figure 5.7
Méthodes de RSSBox


```
</head>
<body>
  <h1>Lire des flux RSS</h1>
```

```
    <!-- Une boite d'information -->
    <div id="rss" class="rss">←❷
      <div>RSS de techno-science.net</div>
      <div id="technoScience"></div>←❸
    </div>
    <script>
// Url de flux RSS 2.0
var technoScienceUrl =
  "http://www.techno-science.net/include/news.xml";
try {
  var technoScienceRSS =
    new RSSBox(technoScienceUrl, "technoScience");←❹
}
catch (exc) {
  alert(exc);
}
infoBox = new InfoBox("rss");←❺
    </script>
  </body>
</html>
```

Nous incluons à partir de la ligne ❶ les trois fichiers JavaScript nécessaires : **RSSBox.js, util.js,** dont il a besoin, et **InfoBox.js,** pour la boîte d'information. La boîte est définie en ❷ par un `div` contenant un `div` d'en-tête, le second contenant le corps à afficher, qui est, dans notre cas, le flux RSS (ligne ❸).

En JavaScript, nous créons les instances de `RSSBox` (ligne ❹) et d'`InfoBox` (ligne ❺) adéquates. Nous plaçons la première dans un `try…catch`, afin de prévenir les rares cas où le navigateur ne serait pas compatible Ajax, le constructeur de `RSSBox` levant alors une exception.

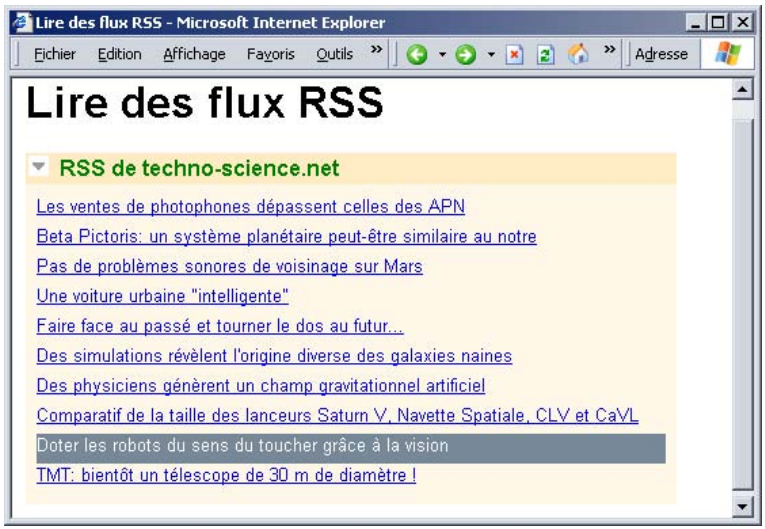
Nous aurions pu aussi bien remplacer la levée de l'exception dans le constructeur par un appel à `Log.error`, ce qui aurait évité de devoir écrire le `try…catch` et aurait réduit l'instanciation à une ligne.

La figure 5.8 illustre le résultat à l'écran.

Nous constatons combien l'inclusion de cette boîte d'information RSS s'écrit avec peu de code HTML (quatre lignes) et JavaScript : une ligne pour créer le lecteur et une ligne pour en faire une boîte d'information. C'est tout le mérite de notre travail par composants, que nous pouvons ensuite assembler en quelques instructions. Le travail est également bien réparti entre ce qui est du ressort du composant, identique dans toutes les utilisations, et ce qui est propre à la page.

Nous pouvons mesurer combien la maintenance des pages en est facilitée.

Figure 5.8
Un flux RSS affiché dans une boîte d'information



En résumé

Après avoir présenté une application fondée sur les données dans la première partie du chapitre, nous avons traité d'une application documentaire. Plusieurs enseignements sont à tirer de ces exemples.

Tout d'abord, nous pouvons mesurer le grand intérêt de RSS, dont la structure assez simple mérite bien son nom de Really Simple Syndication, qui se révèle très utile pour diffuser et intégrer des nouvelles. Concernant ce dernier point, il est aisé de récupérer des nouvelles provenant de plusieurs sources et de les agréger en un seul bloc, grâce à notre mandataire PHP. Nous pourrions ensuite trier, sur le poste client, ces articles par date de parution, source (l'URL du flux) ou catégorie, tous ces éléments figurant dans les flux. Nous pourrions aussi permettre à l'utilisateur de filtrer les résultats dynamiquement.

Nous pouvons ainsi bâtir des applications documentaires véritablement distribuées, du moins en consultation. RSS est à cet égard une réussite éclatante et constitue l'un des piliers de ce qu'on appelle le Web 2.0.

Ces exemples, et tout particulièrement le lecteur RSS, nous ont en outre montré la puissance de structuration qu'offre XML. Nous pouvons créer à partir d'un résultat XML une structure HTML élaborée, sans pour autant nous limiter à un seul rendu HTML, bien au contraire. C'est que XML, comme les bases de données, renferme le contenu et la structure, à l'exclusion de toute représentation. Il est ensuite possible d'en donner de multiples vues, tout comme en bases de données. Cette façon de faire a quelque parenté avec le MVC : le modèle est le flux XML, et la vue est incarnée, entre autres, par les différentes représentations HTML de ces données ou de ces flux.

Le dernier enseignement que nous pouvons tirer de ces exemples est d'ordre technique. Dans ces applications pourtant simples, nous avons pu constater toute la lourdeur de

l'API DOM, à laquelle il manque quelques méthodes ou attributs pratiques, telles que lire ou modifier le contenu d'un élément ou récupérer les enfants d'un type donné. Nous avons été contraints pour cela de créer nous-mêmes ces fonctionnalités ou de les récupérer à partir de bibliothèques, en l'occurrence dojo. Il faut également beaucoup de code pour créer des éléments et leur associer des attributs et du contenu.

Par ailleurs, pour récupérer un élément, nous sommes obligés de naviguer dans l'arbre pas à pas, avec une instruction pour chaque pas. Tout cela est pénible, et c'est pourquoi d'autres techniques ont été mises au point.

E4X (ECMAScript for XML), la plus prometteuse d'entre elles, simplifie considérablement le code, en faisant de XML un type de base du langage JavaScript. Malheureusement, elle n'est supportée que par Firefox 1.5 et par ActionScript 3, le JavaScript de Flash, et son support dans IE-7 n'est pas prévu. Une description claire et concise de ce standard ECMA est disponible à l'adresse *http://developer.mozilla.org/presentations/xtech2005/e4x*.

Deux autres technologies, XPath et XSLT, apparues en 1999, sont beaucoup plus répandues. Elles sont disponibles en JavaScript dans IE-6 et plus et dans Mozilla (Firefox 1.0 et plus), qui représentent vraisemblablement 95 % du parc des navigateurs actuels. Elles sont donc à considérer avec sérieux.

XSLT (eXtensible Stylesheet Language Transformations)

XSLT (eXtended Stylesheet Language Transformations), ou transformations XSL, sert à transformer un document XML en un autre document XML, ou en un document texte, HTML ou encore PDF, sachant que, dans ce dernier cas, il faut utiliser en plus une autre norme, XSL-FO (Formating Output).

Comme RSS, XSLT définit un jeu de balises et d'attributs, avec leurs règles d'imbrication. Dans le cas de RSS, les balises servent à structurer un flux d'articles ; dans celui de XSLT, elles permettent de spécifier une transformation à opérer sur un document source.

Les transformations XSLT sont disponibles aujourd'hui dans tous les navigateurs récents (IE-6, Firefox 1.0 et plus, Netscape 7 et plus, Safari 2 et plus, Opera 8 et plus), quand elles sont associées au document dans le document XML lui-même, à travers une instruction de traitement. Par exemple, le flux RSS du site du journal *Libération,* consultable à l'adresse *http://www.liberation.fr/rss.php,* affiche le résultat HTML de la transformation, alors que le navigateur a reçu un fichier RSS.

Voici le début de ce fichier RSS :

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet type="text/xsl" href="/rss/rss.xsl" media="screen"?>❷
<rss version="2.0">
  <channel>
```

C'est en ligne ❷, juste après la déclaration XML, qu'est spécifiée la transformation à effectuer. Elle est déclarée comme une feuille de style, de type `text/xsl` et non `text/css`, si bien que le navigateur sait quel traitement effectuer.