

Bien développer pour le Web 2.0

Bonnes pratiques Ajax

Prototype • Script.aculo.us • Accessibilité • JavaScript • DOM • XHTML/Css

2^e édition

Christophe Porteneuve

Préface de Tristan Nitot,
président de Mozilla Europe



EYROLLES

Bien développer pour le Web 2.0

CHEZ LE MÊME ÉDITEUR

Ouvrages sur le développement web

E. SLOÏM. – **Mémento Sites web.** *Les bonnes pratiques.*
N°12101, 2007, 14 pages.

A. BOUCHER. – **Ergonomie web.** *Pour des sites web efficaces.*
N°12158, 2007, 426 pages.

R. GOETTER. – **CSS 2 : pratique du design web.**
N°11976, 2^e édition, 2007, 324 pages (Collection Blanche).

M. NEBRA. – **Réussir son site web avec XHTML et CSS.**
N°12307, 2^e édition, 2008, 336 pages.

F. DRAILLARD – **Premiers pas en CSS et XHTML.**
N°12390, 2^e édition 2008, 250 pages.

O. ANDRIEU. – **Réussir son référencement web.**
N°12264, 2008, 302 pages.

A. CLARKE. – **Transcender CSS.** *Sublimez le design web !*
N°12107, 2007, 370 pages.

J.-M. DEFRENCE. – **Premières applications Web 2.0 avec Ajax et PHP.**
N°12090, 2008, 450 pages (Collection Blanche).

K. DJAFAAR. – **Développement JEE 5 avec Eclipse Europa.**
N°12061, 2008, 380 pages (Collection Blanche).

S. BORDAGE, D. THÉVENON, L. DUPAQUIER, F. BROUSSE. – **Conduite de projet Web.** *60 modèles de livrables prêts à l'emploi. Un outil de création de business plan. 3 études de cas.*
N°12325, 4^e édition, 2008, 408 pages.

A. TASSO. – **Apprendre à programmer en Actionscript.**
N°12199, 2007, 438 pages (Collection Noire).

E. PUYBARET. – **Java 1.4 et 5.0.** (coll. *Cahiers du programmeur*)
N°11916, 3^e édition 2006, 400 pages

S. POWERS. – **Débuter en JavaScript.**
N°12093, 2007, 386 pages (Collection Blanche).

T. TEMPLIER, A. GOUGEON. – **JavaScript pour le Web 2.0.**
N°12009, 2007, 492 pages (Collection Blanche).

D. THOMAS *et al.* – **Ruby on Rails.**
N°12079, 2^e édition, 2007, 800 pages (Collection Blanche).

E. DASPET et C. PIERRE DE GEYER. – **PHP 5 avancé.** *De PHP 5.3 à PHP 6.*
N°12369, 5^e édition, 2008, 804 pages (Collection Blanche).

D. SÉGUY, P. GAMACHE. – **Sécurité PHP 5 et MySQL.**
N°12114, 2007, 240 pages (Collection Blanche).

P. ROQUES. – **UML 2. Modéliser une application web.**
N°12389, 4^e édition, 2008, 236 pages (Cahiers du programmeur).

T. ZIADÉ. – **Programmation Python.**
N°11677, 2006, 530 pages (Collection Blanche).

R. RIMELÉ. – **Mémento MySQL.**
N°12012, 2007, 14 pages.

R. GOETTER. – **Mémento CSS.**
N°11726, 2006, 14 pages.

R. GOETTER. – **Mémento XHTML.**
N°11955, 2006, 14 pages.

Autres ouvrages : Web et logiciel libre

D. MERCER, adapté par S. BURIEL. – **Créer son site e-commerce avec osCommerce.**
N°11932, 2007, 460 pages.

A.-L. QUATRAVAUX et D. QUATRAVAUX. – **Réussir un site web d'association... avec des outils libres !**
N°12000, 2^e édition, 2007, 372 pages.

PERLINE, A.-L. et D. QUATRAVAUX, M.-M. MAUDET. – **SPIP 1.9. Créer son site avec des outils libres.**
N°12002, 2^e édition, 2007, 376 pages.

J BATTELLE, trad. D. RUEFF, avec la contribution de S. BLONDEEL – **La révolution Google**
N°11903, 2006, 280 pages.

C. GÉMY. – **Gimp 2.4. efficace.** Dessin et retouche photo.
N°12152, 2008, 402 pages avec CD-Rom.

S. CROZAT. – **Scenari – La chaîne éditoriale libre.**
N°12150, 2007, 200 pages.

I. BARZILAI. – **Mise en page avec OpenOffice.org Writer.**
De la conception à la réalisation.
N°12149, 2007, 338 pages.

S. GAUTIER, C. HARDY, F. LABBE, M. PINQUIER. – **OpenOffice.org 2.2 efficace.**
N°12166, 2007, 420 pages avec CD-Rom (Accès Libre).

Bien développer pour le **Web 2.0**

AJAX • Prototype • Script.aculo.us
XHTML/Css • JavaScript • DOM

2^e édition

Christophe Porteneuve

Préface de Tristan Nitot,
président de Mozilla Europe

EYROLLES



ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com



Le code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2007, 2008, ISBN : 978-2-212-12391-3

Je dédie cette deuxième édition aux membres et fondateurs de l'association Paris-Web et à ceux d'Open Web, qui font tout leur possible, depuis plusieurs années, pour amener toujours davantage de professionnels à placer la qualité, l'accessibilité, l'interopérabilité et la conformité aux standards au cœur de leurs projets web.

Je la dédie aussi aux rédacteurs de Pompage.net, qui travaillent, eux, à permettre aux développeurs web francophones de bénéficier des meilleures ressources web étrangères sur ces sujets.

Adrien Leygues, Antoine Cailliau, Cathy Weber, Cédric Magnin, Denis Boudreau, Éric Daspet, Élie Sloïm, Emmanuel Clément, Fabien Basmaison, Fabrice Bonny, Florian Hatat, François Nonnenmacher, Frédéric Chotard, Frédéric Valentin, Goulven Champenois, Ismaël Touama, Jean-Denis Vauguet, Julien Wajsberg, Laurence Jacquet, Laurent Denis, Laurent Jouanneau, Leo Ludwig, Marc Thierry, Marie Alhomme, Matthieu Pillard, Maurice Svay, Nicolas Gallet, Nicolas Hoffmann, Olivier Gendrin, Olivier Meunier, Pascale Lambert-Charreteur, Pierre Equoy, Renaud Gaudin, Samuel Latchman, Stéphane Deschamps, Stéphanie Booth, Steve Frécinaux, Sylvain Lelièvre, Tristan Nitot, Vincent Valentin, et ceux que je risque d'avoir oubliés : vous êtes formidables !

Préface

Vous tenez donc entre vos mains un exemplaire du livre *Bien développer pour le Web 2.0*. On pourrait croire que ce qui est important dans le titre, c'est « Web 2.0 ». Certes, c'est bien le cas. La participation grandissante des utilisateurs, qui est l'une des deux particularités du Web 2.0, est importante. Vitale, même. Paradoxalement, cette notion d'un Web où chacun pourrait à la fois lire et écrire, consommer et produire, est celle de son inventeur, Tim Berners-Lee, même si peu d'internautes ont réalisé cela.

Mais ce qui est surtout important dans le titre de cet ouvrage, c'est « Bien développer ». Développer « comme il faut ». Car le Web dit « 1.0 » ne s'est pas seulement traduit par un Web où seuls quelques auteurs publiaient pour une foule grandissante de lecteurs : le Web 1.0 s'est aussi traduit par des errements technologiques qui ont fait que la promesse du Web n'a été tenue que partiellement, dans la mesure où les exclus étaient nombreux. Vous n'utilisez pas tel plug-in ? Ah, dommage ! Vous avez recours à tel navigateur trop moderne ? Tant pis pour vous ! Vous souhaitez consulter le site avec votre téléphone mobile ? Vous devrez attendre de trouver un PC connecté. Vous avez désactivé JavaScript dans votre navigateur pour des raisons de sécurité ? Passez votre chemin ! Vous avez un handicap visuel ou des difficultés pour manipuler une souris ? Navré, le service n'est pas conçu pour vous. Combien de millions de personnes se sont retrouvées confrontées à de tels problèmes du Web 1.0 ? C'est impossible de le dire... Mais ça n'était pas tant le Web qui était en cause que la mauvaise façon dont les sites ont été développés, souvent par faute de formation, de recul sur la technologie, encore toute récente.

Aussi, alors que le Web 2.0 fait tant parler de lui, qu'il convient d'acquérir les compétences techniques pour construire un site utilisant ces technologies, autant apprendre dès le début la bonne façon de faire. La bonne façon, c'est celle qui consiste à utiliser des méthodes permettant de conserver la compatibilité avec un éventail aussi large que possible de navigateurs, d'utilisateurs, de paramétrages, et de connexions.

Le Web 2.0 fait deux promesses explicites : plus de participation des utilisateurs, et des interfaces plus agréables et ergonomiques. Il en est une autre qui est implicite : que les développeurs web apprennent des échecs et difficultés du Web 1.0 pour ne pas les répéter. Pour éviter le bricolage que fut le Web à ses débuts, en passant à l'époque de la maturité et de l'industrialisation, en permettant un accès à tous. C'est en cela que ce livre est important : il ne s'agit pas seulement d'apprendre à « développer pour le Web 2.0 » mais aussi d'apprendre à bien développer pour le Web.

Tristan Nitot
Président de Mozilla Europe

Table des matières

Avant-propos	XXV
À qui s'adresse ce livre ?	XXV
Qu'allez-vous trouver dans ce livre ?	XXVI
Qu'apporte cette deuxième édition ?	XXVII
Les standards du Web	XXVII
De quelles technologies parle-t-on ?	XXVIII
Qui est à la barre, et où va-t-on ?	XXIX
À quoi servent les standards du Web ?	XXXII
Et les navigateurs, qu'en pensent-t-ils ?	XXXIV
Quelques mots sur les dernières versions	XXXVI
Qu'est-ce que le « Web 2.0 » ?	XXXVII
Vue d'ensemble, chapitre par chapitre	XXXIX
Première partie : donner vie aux pages	XXXIX
Deuxième partie : Ajax, ou l'art de chuchoter discrètement	XL
Troisième partie : Parler au reste du monde	XL
Des annexes pour le débutant comme pour l'expert	XLI
Aller plus loin...	XLII
À propos des exemples de code	XLII
Remerciements	XLII
Et pour cette deuxième édition...	XLIII
CHAPITRE 1	
Pourquoi et comment relever le défi du Web 2.0 ?	1
Avant/après : quelques scénarios frappants	1
La saisie assistée : complétion automatique de texte	1
Le chargement à la volée	4
La sauvegarde automatique	10
Bien maîtriser ses outils clefs : XHTML, CSS, JS, DOM et Ajax	11

Faire la part des choses : Ajax, c'est quoi au juste ?	13
Plan d'action pour deux objectifs : méthode et expertise	15

PREMIÈRE PARTIE

Donner vie aux pages 17

CHAPITRE 2

Ne prenez pas JavaScript pour ce qu'il n'est pas 19

Mythes et rumeurs sur JavaScript	20
JavaScript serait une version allégée de Java	20
JavaScript ne serait basé sur aucun standard	20
JavaScript serait lent	21
JavaScript serait un langage jouet, peu puissant	21
S'y retrouver entre JavaScript, EcmaScript, JScript et ActiveScript	22
Tout ce que vous ne soupçonnez pas : les recoins du langage	23
Variables déclarées ou non déclarées ?	23
Types de données	25
Fonctions et valeurs disponibles partout	27
<i>Les mystères de parseFloat et parseInt</i>	28
Rappels sur les structures de contrôle	31
<i>Les grands classiques</i>	31
<i>Labélisation de boucles</i>	31
<i>Employer with, mais avec circonspection</i>	33
Opérateurs méconnus	34
<i>Retour rapide sur les grands classiques</i>	34
<i>Opérateurs plus exotiques</i>	35
<i>Comportements particuliers</i>	36
Prise en charge des exceptions	36
<i>Les types d'exceptions prédefinis</i>	36
<i>Capturer une exception : try/catch</i>	37
<i>Garantir un traitement : finally</i>	39
<i>Lancer sa propre exception : throw</i>	40
Améliorer les objets existants	41
<i>Un peu de théorie sur les langages à prototypes</i>	41
<i>Mise en pratique</i>	42
Arguments des fonctions	46
Le binding des fonctions : mais qui est « this » ?	47
Idiomes intéressants	50
<i>Initialisation et valeur par défaut avec </i>	51

Sélectionner une propriété (donc une méthode) sur condition	52
Tester l'absence d'une propriété dans un objet	52
Fonctions anonymes : jamais new Function !	53
Objets anonymes comme hashes d'options	53
Simuler des espaces de noms	54
« Unobtrusive JavaScript » : bien associer code JS et page web	55
Astuces pour l'écriture du code	57
Déjouer les pièges classiques	57
Améliorer la lisibilité	59
Pour aller plus loin	61
Livres	61
Sites	62
 CHAPITRE 3	
Manipuler dynamiquement la page avec le DOM	63
Pourquoi faut-il maîtriser le DOM ?	64
La pierre angulaire des pages vivantes	64
Maîtriser la base pour utiliser les frameworks	64
Comprendre les détails pour pouvoir déboguer	64
Le DOM et ses niveaux 1, 2 et 3	65
Vue d'ensemble des niveaux	65
Support au sein des principaux navigateurs	66
Les aspects du DOM : HTML, noyau, événements, styles...	67
Maîtriser les concepts : document, noeud, élément, texte et collection	67
Le DOM de votre document : une arborescence d'objets	68
Node	70
Document	74
Element	76
Text	77
NodeList et NamedNodeMap	78
DOMImplementation	80
HTMLDocument	81
HTMLElement	82
Quelques bonnes habitudes	83
Déetecter le niveau de DOM disponible	83
Créer les noeuds dans le bon ordre	84
Ne scripter qu'après que le DOM voulu est construit	85
Ne jamais utiliser d'extension propriétaire	87
Utiliser un inspecteur DOM	87
L'inspecteur DOM de Firefox/Mozilla	87
L'inspecteur DOM de Firebug	90

Répondre aux événements	94
Les truands : les attributs d'événement dans HTML	94
La brute : les propriétés d'événement dans le DOM niveau 0	95
Le bon : addEventListener	96
Accommoder MSIE	98
La propagation : capture ou bouillonnement ?	99
Le modèle le plus courant : le bouillonnement	100
La capture, ou comment jouer les censeurs	101
L'objet Event	102
Récupérer l'élément déclencheur	102
Stopper la propagation	102
Annuler le traitement par défaut	103
JavaScript, événements et accessibilité	103
Besoins fréquents et solutions concrètes	105
Décoration automatique de libellés	106
Validation automatique de formulaires	111
Résoudre les écueils classiques	115
MSIE et la gestion événementielle	115
MSIE et le DOM de select/option	116
Les principaux points problématiques	116
Pour aller plus loin	117
Livres	117
Sites	117

CHAPITRE 4

Prototype : simple, pratique, élégant, portable ! 119

Avant de commencer.....	120
Un mot sur les versions	120
Navigateurs pris en charge	121
Vocabulaire et concepts	121
Espaces de noms et modules	121
Itérateurs	122
Élément étendu	122
Alias	123
Comment utiliser Prototype ?	124
You allez aimer les dollars	124
La fonction \$ facilite l'accès aux éléments	125
La fonction \$A joue sur plusieurs tableaux	126
La fonction \$H, pour créer un Hash	127
La fonction \$F, des valeurs qui sont les vôtres	128
La fonction \$R et les intervalles	128

La fonction \$\$ et les sélecteurs CSS	129
La fonction \$w joue sur les mots	129
Jouer sur les itérations avec \$break et return	130
Extensions aux objets existants	131
Un Object introspectif	131
Gérer correctement le binding	133
Plus de puissance pour les fonctions	135
De drôles de numéros	139
Prototype en a dans le String	140
<i>Suppression de caractères : strip, stripTags, stripScripts, truncate</i>	<i>141</i>
<i>Transformations : camelize, capitalize, dasherize, escapeHTML, gsub,</i>	
<i>interpret, sub, underscore, unescapeHTML</i>	<i>142</i>
<i>Fragments de scripts : extractScripts, evalScripts</i>	<i>144</i>
<i>Conversions et extractions : scan, toQueryParams, parseQuery, toArray,</i>	
<i>inspect, succ</i>	<i>145</i>
<i>Inspection : blank, empty, endsWidth, include, startsWith</i>	<i>146</i>
Des tableaux surpuissants !	147
<i>Conversions : from, inspect</i>	<i>147</i>
<i>Extractions : first, last, indexOf, lastIndexOf</i>	<i>147</i>
<i>Transformations : clear, compact, flatten, without, reverse, reduce, uniq,</i>	
<i>intersect</i>	<i>148</i>
Modules et objets génériques	151
Enumerable, ce héros	151
<i>L'itération elle-même : each</i>	<i>151</i>
<i>Le contexte d'itération</i>	<i>152</i>
<i>Tests sur le contenu : all, every, any, some, include, member, size</i>	<i>153</i>
<i>Extractions : detect, find, filter, findAll, select, grep, max, min, pluck, reject,</i>	
<i>partition</i>	<i>154</i>
<i>Transformations et calculs : collect, map, inGroupsOf, inject, invoke,</i>	
<i>sortBy, zip</i>	<i>156</i>
<i>Conversions : toArray, entries, inspect</i>	<i>159</i>
Il suffit parfois d'un peu de Hash	159
ObjectRange : intervalles d'objets	162
PeriodicalExecuter ne se lasse jamais	162
Vous devriez ré-évaluer vos modèles	163
<i>Utiliser un motif de détection personnalisé</i>	<i>165</i>
Prise en charge de JSON	166
Plus de POO classique : classes et héritage	168
L'héritage et la surcharge de méthode	170
Manipulation d'éléments	171
Element, votre nouveau meilleur ami	171

<i>Element.Methods et les éléments étendus</i>	171
<i>Valeur de retour des méthodes</i>	173
<i>Enfin manipuler librement les attributs : hasAttribute, readAttribute et writeAttribute</i>	173
<i>Élément es-tu là ? hide, show, toggle et visible</i>	174
<i>Gestion du contenu : cleanWhitespace, empty, remove, replace, update, insert, wrap et la syntaxe constructeur</i>	174
<i>Styles et classes : addClassName, getOpacity, getStyle, hasClassName, match, removeClassName, setOpacity, setStyle et toggleClassName</i>	177
<i>Les copains d'abord : ancestors, childElements, descendantOf, descendants, firstDescendant, immediateDescendants, previousSiblings, nextSiblings, siblings</i>	179
<i>Bougez ! adjacent, down, next, previous, select et up</i>	180
<i>Positionnement : absolutize, clonePosition, cumulativeOffset, cumulativeScrollOffset, getDimensions, getHeight, getOffsetParent, getWidth, makePositioned, positionedOffset, relativize, undoPositioned et viewportOffset</i>	182
<i>Défilement et troncature : makeClipping, scrollTo, undoClipping</i>	185
<i>Ajouter des méthodes aux éléments étendus</i>	185
<i>Quelques détails pour finir...</i>	186
La gestion du viewport	187
Selector, l'objet classieux	188
Manipulation de formulaires	190
Field / Form.Element	191
<i>Un mot sur Field.Serializers</i>	192
Form	193
Form.Observer	195
Field.Observer	196
Gestion unifiée des événements	196
Event	197
Observer (ou cesser d'observer) un événement	197
Examiner un événement déclenché	199
<i>Coincer le coupable</i>	199
<i>Étouffer l'affaire</i>	202
<i>Déterminer l'arme du crime</i>	202
Utiliser les événements personnalisés	205
<i>L'événement personnalisé dom:loaded de l'objet document</i>	206
Form.EventObserver	206
Field.EventObserver	207
L'objet global Prototype	207

Pour aller plus loin...	208
Sites	208
Groupe de discussion	208
Canal IRC	209
CHAPITRE 5	
Déboguer vos scripts.....	211
Déboguer d'abord sur Firefox avec Firebug	212
La console : rapide, efficace, et... très sous-utilisée	213
Déboguer des scripts chargés	217
Regarder sous le capot d'Ajax	220
Examiner les performances avec le profileur	221
Peaufiner sur IE6 et IE7	222
L'utopie d'un Windows pour deux	222
Le poids plume : Microsoft Script Debugger	224
<i>Préparer le terrain avec une page à échec garanti</i>	224
<i>Configurer MSIE pour autoriser le débogage</i>	225
<i>Les possibilités du débogueur</i>	227
Le poids lourd : Visual Web Developer 2008 Express Edition	230
Peaufiner sur IE8	236
Peaufiner sur Safari	238
Peaufiner sur Opera	239
Pour aller plus loin	242
Sites	242
DEUXIÈME PARTIE	
Ajax, ou l'art de chuchoter	243
CHAPITRE 6	
Les mains dans le cambouis avec XMLHttpRequest	245
Anatomie d'une conversation Ajax	245
Un petit serveur pour nos tests	246
Installation de Ruby	248
<i>Sous Windows</i>	249
<i>Sous Linux/BSD</i>	250
<i>Sous Mac OS X</i>	251
<i>Un mot sur le cache</i>	251
Un petit serveur HTTP et un code dynamique simple	252

La petite histoire de XMLHttpRequest	254
Origines et historique	254
Bien préparer un échange asynchrone	255
ActiveX versus objet natif JavaScript	255
Créer l'objet requêteur	255
Décrire notre requête	257
Envoyer la requête	258
Recevoir et traiter la réponse	259
Une utilisation complète de notre petit serveur d'exemple	260
Comment surveiller les échanges Ajax de nos pages ?	263
Types de réponse : XHTML, XML, JS, JSON...	265
Bien choisir son type de réponse	266
Une réponse textuelle simple : renvoyer une donnée basique	267
<i>Exemple 1 : sauvegarde automatique</i>	267
<i>Exemple 2 : barre de progression d'un traitement serveur</i>	275
Fragments de page prêts à l'emploi : réponse XHTML	280
Dans la cour des grands : XPath pour traiter des données XML complexes ..	286
<i>Vite et bien : utilisation de DOM niveau 3 XPath</i>	287
<i>En simulant : utilisation de GoogleAJAXSLT</i>	291
Piloter la page en renvoyant du JavaScript	293
JSON : l'idéal pour des données structurées spécifiques	296

CHAPITRE 7

Ajax tout en souplesse avec Prototype	303
Prototype encore à la rescousse	304
Ajax.Request, c'est tellement plus simple !	304
Plus de détails sur Ajax.Request	306
Quelques mots sur les fonctions de rappel	308
Interpréter la réponse avec Ajax.Response	309
Ajax.Updater : mettre à jour un fragment XHTML, exécuter un script	310
Différencier la mise à jour entre succès et échec	313
Presque magique : Ajax.PeriodicalUpdater	314
Comprendre l'option decay	315
Petits secrets supplémentaires	316
Pour aller plus loin...	318
Livres	318
Sites	318
Groupe de discussion	319
Canal IRC	319

CHAPITRE 8

Une ergonomie de rêve avec script.aculo.us.....	321
Une ergonomie haut de gamme avec script.aculo.us	322
Les effets visuels	322
Les effets noyau	323
<i>Invocation de l'effet</i>	323
<i>Options communes à tous les effets noyau</i>	324
<i>Fonctions de rappel</i>	325
<i>Et si on essayait quelques effets ?</i>	325
<i>Les effets combinés</i>	338
<i>Files d'effets</i>	342
Glisser-déplacer	346
<i>Faire glisser un élément avec Draggable</i>	347
<i>Gérer le dépôt d'un élément avec Droppables</i>	352
Tri de listes par glisser-déplacer	362
<i>Que peut-on trier ?</i>	363
<i>Activer les fonctions d'ordonnancement</i>	363
<i>Désactiver l'ordonnancement</i>	372
<i>Envoyer l'ordre au serveur</i>	372
Complétion automatique de texte	373
<i>Création d'un champ de saisie à complétion automatique</i>	373
<i>Interaction clavier et souris</i>	375
<i>Un premier exemple</i>	376
<i>Personnalisation des contenus renvoyés</i>	382
Et ce n'est pas tout ! Il y a d'autres services	390
Avoir du recul : les cas où Ajax est une mauvaise idée	390
Ajax et l'accessibilité	391
<i>Considérations techniques</i>	391
<i>Considérations ergonomiques</i>	392
Utilisations pertinentes et non pertinentes	393
Pratiques recommandées	394
<i>Principes généraux</i>	394
<i>Ergonomie et attentes de l'utilisateur</i>	395
<i>Cognitif/Lecteurs d'écran</i>	396
Pour aller plus loin...	397
Site	397
Groupe de discussion	397
Canal IRC	397

TROISIÈME PARTIE

Interagir avec le reste du monde 399

CHAPITRE 9

Services web et REST : nous ne sommes plus seuls 401

Pourquoi la page ne parlerait-elle qu'à son propre site ?	402
Contraintes de sécurité sur le navigateur	402
Une « couche proxy » sur votre serveur	403
Architecture de nos exemples	404
Comprendre les services web	405
Qu'est-ce qu'une API REST ?	406
Cherchons des livres sur Amazon.fr	407
Obtenir une clef pour utiliser l'API	408
L'appel REST à Amazon.fr	409
<i>Anatomie de la requête</i>	409
<i>Le document XML de réponse</i>	411
Notre formulaire de recherche	411
Passer par Ajax	414
<i>La couche serveur, intermédiaire de téléchargement</i>	414
<i>Intercepter le formulaire</i>	416
De XML à XHTML : la transformation XSLT	418
<i>Notre feuille XSLT</i>	418
<i>Apprendre XSLT à notre page</i>	422
<i>Charger la feuille XSLT au démarrage</i>	422
<i>Effectuer la transformation</i>	423
<i>Embellir le résultat</i>	425
Rhumatismes 2.0 : prévisions météo	430
Préparer le projet	431
Récupérer les prévisions d'un lieu	433
<i>Requête et réponse REST</i>	433
<i>Initialisation de la page et obtention des prévisions</i>	436
<i>Et la feuille XSLT ?</i>	439
<i>Les petites touches finales</i>	443
Rechercher un lieu	447
<i>Préparons le terrain</i>	448
<i>Éblouissez vos amis avec Ajax.XSLTCompleter</i>	451
<i>Brancher les composants ensemble</i>	453
Gérer des images chez Flickr	456
Obtenir une clef API chez Flickr	457
Format général des requêtes et réponses REST chez Flickr	458

Préparer le terrain	459
Chargement centralisé parallèle des feuilles XSLT	461
Obtenir les informations du jeu de photos	463
Récupérer les photos du jeu	471
Afficher une photo et ses informations	477
Pour aller plus loin...	483
 CHAPITRE 10	
L'information à la carte : flux RSS et Atom	485
Aperçu des formats	486
Une histoire mouvementée	486
<i>RSS 0.9x et 2.0 : les « bébés » de Dave Winer</i>	486
<i>RSS 1.0 : une approche radicalement différente</i>	486
<i>Atom, le fruit de la maturité</i>	487
Informations génériques	488
Le casse-tête du contenu HTML	488
Récupérer et afficher un flux RSS 2.0	489
Format du flux	490
Préparer le terrain	491
<i>La feuille XSLT</i>	493
Chargement et formatage du flux	495
Ajustements des dates et titres	498
Affichage plus avancé et flux Atom 1.0	499
Notre flux Atom	500
Préparation de notre lecteur de flux	502
La feuille XSLT et le formatage	504
<i>Charger la feuille et le flux</i>	506
<i>Afficher dynamiquement les billets complets</i>	509
Les mains dans le cambouis : interpréter le HTML encodé	512
<i>Traiter des quantités massives de HTML encodé</i>	512
<i>Les dates W3DTF</i>	514
Pour aller plus loin...	516
Livres	516
Sites	517
 CHAPITRE 11	
Mashups et API 100 % JavaScript.....	519
Des cartes avec Google Maps	520
Exigences techniques de l'API	520
Créer une carte	522
Ajouter des contrôles et des marqueurs	524

<i>Des contrôles pour bouger et s'y retrouver</i>	524
<i>Des marqueurs pour identifier des points spécifiques</i>	525
Et tellement d'autres choses...	530
Et si l'on veut d'autres types de cartes ?	530
Des graphiques avec Google Chart	531
Le principe	531
Créer un premier graphique	532
Les grands axes de personnalisation	534
<i>La taille</i>	534
<i>Les données</i>	534
<i>Le type de graphique</i>	539
<i>Les couleurs</i>	541
<i>Mais aussi...</i>	542
Carte ou graphique ? Les cartographies	543
Alternatives suivant les besoins	545
Pour aller plus loin	547
Sites	547

ANNEXE A

Bien baliser votre contenu : XHTML sémantique	551
Les avantages insoupçonnés	552
Pour le site et ses propriétaires	552
Pour le développeur web	553
Règles syntaxiques et sémantiques	553
La DTD et le prologue	555
XHTML, oui mais lequel ?	556
Le balisage sémantique	556
Les balises XHTML 1 Strict par catégorie	557
Balises structurelles	557
Balises sémantiques	558
Balises de liaison	560
Balises de métadonnées	561
Balises de présentation	561
Balises de formulaires et d'interaction	563
Balises dépréciées	564
Attributs incontournables	565
Besoins fréquents, solutions concrètes	566
Un formulaire complexe mais impeccable	566
Un tableau de données à en-têtes groupés	573
Un didacticiel technique	575

Pour aller plus loin...	578
Livres	578
Sites	578
 ANNEXE B	
Aspect irréprochable et flexible : CSS 2.1	581
Statut, état et vocabulaire	582
Les versions de CSS	582
Prise en charge actuelle	582
Le jargon CSS	583
Bien comprendre la « cascade » et l'héritage	584
Le sens de la cascade	584
Calcul de la spécificité d'une règle	585
Que se passe-t-il avec la présentation dans HTML ?	586
L'héritage	586
<i>De la valeur spécifiée à la valeur concrète</i>	587
Les modèles de boîte et de mise en forme visuelle	587
Les côtés d'une boîte : ordre et syntaxes courtes	588
Unités absolues et relatives	588
Marge, bordure et espacement	590
Éléments en ligne et de type bloc	591
Éléments remplacés	591
Fusion des marges	591
Le modèle W3C et le modèle Microsoft	592
Tour d'horizon des sélecteurs	593
Groupement	594
Pseudo-éléments	594
Tour d'horizon des propriétés	595
De l'art de réaliser des CSS légères	595
Propriétés du modèle de boîte : marges, espacements et bordures	596
Propriétés de formatage visuel : positionnement, largeur, hauteur, baseline	596
Propriétés de contenu généré automatiquement	597
Propriétés de pagination	598
Propriétés de couleurs et d'arrière-plan	599
Propriétés de gestion de la police de caractères	599
<i>Taille de police</i>	600
<i>Famille de polices</i>	600
<i>Tout spécifier d'un coup !</i>	600
Propriétés de gestion du corps du texte	602
<i>L'espacement dans le corps du texte</i>	602
Propriétés des tableaux	603

Propriétés de l'interface utilisateur	603
Pour aller plus loin...	604
Livres	604
Sites	604

ANNEXE C

Le « plus » de l'expert : savoir lire une spécification 605

De l'intérêt d'aller chercher l'information à la source	606
Certitude et précision	606
« On m'a dit que là-dessus, c'est toi qui sais tout » : l'expertise	606
Les principaux formats de spécifications web	607
Les recommandations du W3C	607
Les grammaires formelles de langages à balises : DTD et schémas XML	607
Les RFC de l'IETF : protocoles et formats d'Internet	608
S'y retrouver dans une recommandation W3C	608
URL et raccourcis	608
Structure générale d'une recommandation	609
Recours à des syntaxes formelles	613
Les descriptions de propriétés CSS	615
Les descriptions de propriétés et méthodes DOM	616
Déchiffrer une DTD	618
Naviguer dans un schéma XML	622
Parcourir une RFC	625
Format général d'une RFC	625
Structure générale d'une RFC	627
Vos spécifications phares	629

ANNEXE D

Développer avec son navigateur web 631

Le cache peut être votre pire ennemi	632
Le rafraîchissement strict	632
Vider le cache	632
Configurer le cache	633
Firefox, favori du développeur grâce aux extensions	634
Firebug	634
Web Developer Toolbar	636
Les trésors du menu Développement caché dans Safari	639
MSIE 6-7 et la Internet Explorer Developer Toolbar	646
MSIE 8b1 et son outil de développement intégré	647
Opera et ses nouveaux outils de développement	650

ANNEXE E

Tour d'horizon des autres frameworks JavaScript	653
jQuery	653
YUI (Yahoo! User Interface)	654
Dōjō	655
Ext JS	656
Base2	657
MooTools	658
Et deux autres, très liés à la couche serveur...	659
GWT	659
ASP.NET AJAX (ex-Atlas)	660
Petit tableau récapitulatif	661
La vraie fausse question de la taille...	661
Index.....	663

Avant-propos

Avant d'entrer dans le vif du sujet, faisons le tour des domaines abordés par ce livre, ce qu'il contient et ce que j'ai choisi d'omettre. Nous verrons comme le livre articule ses différents thèmes pour en faciliter l'apprentissage.

À qui s'adresse ce livre ?

Toute personne qui s'intéresse de près ou de loin aux technologies web trouvera une utilité à cet ouvrage. Précisons néanmoins qu'*une connaissance préalable des technologies de contenu web statique est préférable* : en l'occurrence, HTML (idéalement XHTML) et CSS.

Il est également recommandé d'avoir déjà utilisé un langage de programmation impératif, tel que C, C++, Delphi, Java... Histoire de connaître les notions de variable, fonction, boucle, tableau, etc.

Dans l'idéal, vos connaissances sont à jour, donc conformes aux standards (XHTML strict, CSS 2.1), et solides, notamment en termes de balisage sémantique. Les lecteurs ayant des lacunes sur ces technologies pourront toutefois trouver une présentation succincte des principes fondamentaux dans les annexes A et B, ainsi que de nombreuses ressources — papier ou en ligne — pour parfaire leurs connaissances, dans la bibliographie de ces annexes.

Il n'est par ailleurs pas nécessaire d'avoir des compétences préalables en JavaScript ou DOM, ces sujets étant présentés en détail dans ces pages. En somme, ce livre trouvera son public tant auprès des professionnels chevronnés désireux de se mettre à jour, que des étudiants souhaitant aller au-delà de leurs cours de technologies web, souvent sommaires et trop empiriques, voire obsolètes.

Qu'allez-vous trouver dans ce livre ?

Le livre est découpé en trois parties, précédées de cet avant-propos et d'un chapitre introductif qui présente le Web 2.0 et ses technologies.

Par ailleurs, l'ouvrage est développé sur deux axes forts : un axe thématique et un axe méthodologique et qualitatif. Le premier axe guide le plan, tandis que le second est transversal :

- La **première partie** présente en détail les technologies qui font *vivre* la page web elle-même. Elles sont trop souvent mal connues : JavaScript, DOM et, pour gagner en agilité et en puissance, l'excellente bibliothèque Prototype.
- La **deuxième partie** explore ce qui fait réellement Ajax, à savoir l'objet XMLHttpRequest, moteur de requêtes asynchrones, et les frameworks déjà établis dans l'univers du Web 2.0, notamment Prototype et script.aculo.us.
- La **troisième partie** pousse la réflexion et l'utilisation plus loin en ouvrant vos pages sur des contenus et services externes, au travers des Web Services, des API REST et des flux de syndication aux formats RSS et Atom.

L'ouvrage est complété par cinq annexes :

- Les **annexes A et B** fournissent les bases des deux principales technologies de contenu : XHTML et CSS, dans leurs versions récentes.
- L'**annexe C** constitue un plus indéniable : en vous expliquant clairement comment exploiter au mieux les documents de référence qui font le Web (RFC, DTD, recommandations W3C...), elle vous donne les clés d'une connaissance actuelle et faisant autorité.
- L'**annexe D**, elle, est plutôt à lire d'entrée de jeu : elle donne les clés d'un développement plus productif dans votre navigateur, et vous évite de peiner avec les questions de cache en manipulant les exemples de ce livre.
- L'**annexe E** termine avec un tour d'horizon des principaux *frameworks* Java Script, autre Prototype et script.aculo.us, en tentant d'en éliminer rapidement les forces et les faiblesses, et de donner les clés d'un choix pertinent.

Dans tous ces chapitres, j'ai tenté d'insuffler au lecteur le souci constant de la qualité, tant pour la technique elle-même que pour la méthodologie de travail. Qu'il s'agisse d'*unobtrusive JavaScript* (concept que nous étudierons en détail au chapitre 2), de balisage sémantique, de CSS efficaces, d'accessibilité, ou du bon choix de format pour un flux de syndication ou le résultat d'une requête Ajax, j'ai fait de mon mieux pour donner les clés d'un savoir-faire haut de gamme, constituant un avantage compétitif certain, à l'heure où tout un chacun n'hésite pas à clamer sur son CV qu'il est un « développeur web expert ».

Adresse au lecteur

Je me suis posé la question du pronom employé pour désigner l'auteur. D'aucuns diront que le « nous », d'usage, est plus modeste... Peut-être, mais j'écris ce livre pour vous, pour qu'il vous soit utile. Et s'il ne vous plaît pas, c'est *ma* faute, pas *la nôtre*. Alors, pour faire plus simple et plus convivial, ce sera « je ».

Qu'apporte cette deuxième édition ?

Presque deux ans après la première édition de cet ouvrage, il est temps de remettre à jour le contenu, bien sûr, mais aussi d'y effectuer les ajouts et amendements qui font tout le sel d'une nouvelle édition. Au-delà des corrections de coquilles qui avaient échappé à notre vigilance, voici l'essentiel des différences entre la première et la deuxième édition :

- *des clarifications* partout où cela nous a semblé nécessaire, que ce soit lors de nos relectures ou suite à des commentaires envoyés par des lecteurs ;
- *une mise à jour complète du contenu* commandée par l'évolution de l'état de l'art : nouveaux standards, nouveaux outils, nouvelles versions de navigateurs, de bibliothèques (notamment les nombreuses nouveautés de Prototype 1.6) et autres technologies ;
- *un nouveau chapitre crucial* : le **chapitre 5, « Déboguer votre JavaScript »**. Ayant moi-même l'expérience quotidienne de l'écriture de scripts avancés, j'ai constaté que la quasi-totalité des ressources en ligne ou papier étaient lacunaires sur ce sujet. Ce chapitre tente d'apporter une méthodologie et des connaissances techniques pour faciliter la mise au point de scripts JavaScript.
- *un nouveau chapitre* sur les « **Mashups et API 100 % JavaScript** », qui illustre la tendance désormais répandue d'utiliser directement côté client des services tiers. Nous verrons d'abord l'intégration d'une géolocalisation *via* le célèbre Google Maps, puis avec le tout récent et si sympathique Google Charts.

Les standards du Web

Tout le monde en parle, tout le monde dit que c'est bien et qu'il faut les respecter. Mais personne ne dit vraiment de quoi il s'agit, pourquoi c'est mieux, et vers où se tourner pour mettre le pied à l'étrier.

À l'heure où une portion significative des développeurs web français chevonnés, et la majorité des jeunes diplômés français croyant « connaître le développement web », ignorent ce qu'est le W3C, ne savent pas donner la dernière version de HTML, sont

dans le flou sur les différences exactes entre XHTML et HTML, et pensent que CSS se résume à coller des balises `div` et des attributs `class` et `style` partout, nous avons toujours du chemin à faire en termes d'évangélisation et d'éducation en général.

De quelles technologies parle-t-on ?

Commençons par passer en revue les technologies qui font aujourd'hui figure de standards du Web. Je restreindrai la liste aux technologies qui se rapprochent de notre propos, sous peine d'y consacrer de nombreuses pages.

- **HTML** (*HyperText Markup Language*) est le langage établi de description de contenu dans une page web. Dérivé du SGML, sa syntaxe est un peu trop permissive pour éviter toute ambiguïté et permettre un traitement automatisé vraiment efficace.
- **XML** (*eXtensible Markup Language*) est une syntaxe plus formelle de balisage de contenu, qui garantit le traitement automatique du document sans risquer ni ambiguïtés, ni soucis de jeux de caractères, ni limitations de types ou tailles de contenu.
- **XHTML** revient essentiellement à appliquer à HTML les contraintes syntaxiques de XML, ouvrant ainsi la porte au traitement fiable du contenu des pages web.
- **CSS** (*Cascading Style Sheets*, généralement juste « feuilles de style » en français) est une technologie de présentation permettant une mise en forme extrêmement avancée des contenus compatibles XML (et, par souci de flexibilité, de HTML aussi). Les possibilités sont énormes et vont bien au-delà de ce que permettaient les quelques balises « présentation » de HTML.
- **DOM** (*Document Object Model*) décrit une série d'outils à destination des programmeurs (on parle d'*interfaces*) permettant de représenter et de manipuler en mémoire un document compatible XML. Ces manipulations sont pratiquement illimitées et constituent un des piliers d'une page web « vivante ». Parmi les sous-parties de DOM, on citera notamment *Core*, qui fournit le noyau commun à tous les types de documents ; *HTML*, spécialisé dans les pages web ; et enfin *Events*, qui gouverne le traitement des événements associés aux éléments du document.
- **JavaScript** est un langage de script, dynamique, orienté objet et disposant de nombreuses fonctions avancées, aujourd'hui disponible sous une forme ou sous une autre dans tous les navigateurs un tant soit peu répandus. Sans lui, pas de pages vivantes, pas de Web 2.0, pas d'Ajax !
- **XMLHttpRequest** est un objet capable d'envoyer des requêtes asynchrones *via* HTTP (voilà une phrase qui ne vous dit peut-être pas grand-chose ; pas d'inquiétude, le chapitre 6 vous éclairera bientôt). Utilisé en JavaScript, il constitue le cœur d'Ajax.

- **RSS** 1.0 (*RDF Site Summary*) est un format de flux de syndication, défini de façon beaucoup plus formelle que ses homonymes de versions 0.9x ou 2.0 (où l'abréviation signifie *Really Simple Syndication*), lesquels sont plus répandus mais moins puissants. Il est basé sur **RDF** (*Resource Description Framework*), une grammaire formelle de représentation de la connaissance, autour de laquelle gravite l'univers du Web sémantique (pour plus de détails sur le sujet, consultez par exemple <http://www.w3.org/2001/sw/>).
- **Atom** est le format de flux de syndication le plus récent, sans doute le plus puissant et le plus efficace aussi, sans pour autant verser dans la complexité.

Parmi les standards du Web, on trouve encore de nombreuses technologies très employées, comme PNG (format d'image), SOAP et les Web Services, XSL et XSLT ; ainsi que d'autres encore trop rarement employées, par exemple SVG (images vectorielles), MathML (formules mathématiques), SMIL (multimédia)...

Qui est à la barre, et où va-t-on ?

Ces standards ne s'inventent pas seuls ; à leur origine, on trouve le plus souvent une organisation, un comité ou une association, parfois une entreprise, plus rarement encore un individu. Mais ceux qui ont fait naître une technologie n'en assurent pas toujours bien l'évolution, comme c'est le cas pour HTML par exemple.

Comprendre qui s'occupe d'un standard permet de savoir où suivre son évolution, de déterminer à quoi s'attendre dans les années à venir, et de mieux comprendre ses orientations et les choix qui les gouvernent.

- **(X)HTML** a été principalement maintenu par le **W3C** (*World Wide Web Consortium*), un groupement international d'associations, d'entreprises et d'individus en charge de la plupart des technologies du Web, principalement dans le contexte des navigateurs. Hélas, après avoir sorti HTML 4.01 en 1999, le W3C a délaissé HTML pour se concentrer sur le Web sémantique, CSS et les technologies autour d'XML.

Or le problème est que HTML est l'outil fondamental de tout développeur web, quelle que soit la technologie côté serveur, qu'on soit Ajax ou non, Web 2.0 ou non. Figé jusqu'au début du siècle, HTML 4.01 échouait à satisfaire nombre de besoins récurrents.

Devant la difficulté à remobiliser le W3C autour de HTML, un groupement séparé a vu le jour, le **WHAT WG** (*Web Hypertext Application Technology Working Group*, <http://whatwg.org>). Constitué principalement de figures de proue des standards, presque tous par ailleurs membres du W3C, il jouit déjà d'une excellente notoriété et d'une large approbation. Il vise à mettre au point plusieurs standards, dont deux souvent désignés par dérision sous l'appellation commune « HTML 5 » :

Web Applications 1.0 et Web Forms 2.0. Ces deux projets augmentent énormément les possibilités pour le développeur web, et plusieurs navigateurs de premier plan ont annoncé leur intention de les prendre en charge...

Pour finir, c'est le W3C qui a repris la main en 2007, et son groupe de travail HTML est reparti sur les bases du HTML 5, développé par le WHAT WG. La prise en charge est prometteuse dans les dernières versions des navigateurs. À surveiller attentivement, donc !

- **CSS** est également l'œuvre du **W3C**, dont il reste un cheval de bataille important. Depuis la version 2, remontant à 1998 (!), le standard évolue de façon double. D'un côté, une version 2.1 est en chantier permanent (la dernière révision date de juillet 2007) et constitue une sorte de correction de la version 2, qui en précise les points ambigus, ajoute quelques compléments d'information, etc. De l'autre, la version 3 est un chantier proprement pharaonique, à tel point que le standard est découpé en pas moins de 37 modules. Parmi ceux-là, certains font l'objet de beaucoup d'attentions, et sont au stade de la *recommandation candidate* (dernière étape avant l'adoubement au rang de standard), ou de *dernier appel à commentaires*. Il ne s'agit au total que de 11 modules sur 37. Pour les autres, soit le travail n'a carrément pas démarré, soit ils disposent d'une ébauche qui, parfois, stagne pendant des années (le module de gestion des colonnes, pourtant réclamé à corps et à cris par beaucoup, a ainsi gelé entre janvier 2001 et décembre 2005, et sa dernière ébauche remonte à plus d'un an !). Enfin, même si CSS a d'ores et déjà révolutionné la création de pages web, on verra qu'il existe un bel écart entre les dernières versions et l'état de l'art dans les navigateurs...
- **DOM** est aussi à mettre au crédit du **W3C**. En DOM, on ne parle pas de versions mais de *niveaux*. Le W3C travaille régulièrement dessus au travers de ses sous-projets : *Core*, *HTML*, *Events*, *Style*, *Views* et *Traversal and Range*.
- **JavaScript** a été inventé en 1995 par **Brendan Eich** pour le navigateur Netscape Navigator 2.0. C'est aujourd'hui l'**ECMA**, un organisme international de spécifications, qui gère son évolution au travers des diverses éditions du standard ECMA-262. Brendan Eich continue à piloter la technologie, *via* les travaux actuels sur ES4, c'est-à-dire JavaScript 2.0, et travaille comme directeur technique de Mozilla.
- **XMLHttpRequest** a été inventé par **Microsoft** pour Internet Explorer (MSIE) 5.0. Depuis 2002, des équivalents ont fait leur apparition dans la plupart des navigateurs, au point qu'un standard **W3C** est en cours de rédaction (depuis plus de deux ans...) pour enfin ouvrir totalement la technologie.
- **RSS** est un sigle qui masque en réalité deux technologies bien distinctes. La première version historique, la 0.90, vient de Netscape (1999). Les versions 0.9x

suivantes et 2.0 sont l'œuvre de Dave Winer tout seul dans son coin, et fournissent une solution simple (et même simpliste) aux besoins les plus courants de la syndication de contenu. Il s'agit de standards gelés, qui n'évolueront plus. La version 1.0 est beaucoup plus puissante, mais aussi plus complexe, car basée sur RDF, donc sur un standard formel lourd réalisé par le **W3C**. Elle est encore, pour l'instant, moins utilisée que ses homonymes.

- **Atom** a été défini, contrairement à RSS, dans la stricte tradition des standards Internet : au moyen d'un forum ouvert de discussion, et encadré dès le début par l'**IETF**, organisme international chargé de la plupart des protocoles Internet (comme HTTP). Il gagne sans cesse en popularité, et constitue très officiellement un standard (ce qu'on appelle une RFC) depuis décembre 2005, sous le numéro 4287 (<http://tools.ietf.org/html/rfc4287>).

Les principaux acteurs des standards du Web sont donc le W3C et, sans doute de façon moins visible pour l'utilisateur final, l'IETF. On constate néanmoins que le premier est parfois prisonnier de sa propre bureaucratie, au point que des groupes externes reprennent parfois le flambeau, comme cela a été le cas autour de HTML avec le WHAT WG.

Ce panorama ne serait pas complet sans évoquer le WaSP (*Web Standards Project*, <http://webstandards.org>), véritable coalition d'individus ayant appréhendé tout l'intérêt des standards du Web et la portée de leur application. Ce groupe fut un acteur important de l'arrêt de la « guerre des navigateurs » qui fit rage dans les années 1990, laissant Navigator sur le carreau et faisant entrer MSIE dans la léthargie qu'on lui a longtemps connue.

Mais surtout, il œuvre sans relâche pour rallier toujours plus d'acteurs, notamment les éditeurs commerciaux, à la prise en charge des standards. En collaborant avec Microsoft, mais aussi Adobe et Macromedia (du temps où ils n'avaient pas fusionné), ainsi que de nombreux autres, le WaSP aide à rendre les produits phares du marché plus compatibles avec les standards et l'accessibilité. Vraiment, grâce à leur soient rendues ! Sans eux, on en serait encore à devoir annoter la moindre mention technique dans un ouvrage comme celui-ci à coups de « IE seulement », « NN seulement », « non supporté », etc.

Quels sont donc ces avantages extraordinaires qui ont convaincu tant de volontaires de fonder ou rejoindre le WaSP, et de partir en croisade auprès des éditeurs ? Quelles perspectives ensoleillées ont-ils vues ? C'est ce que je vais vous expliquer dans la section suivante.

À quoi servent les standards du Web ?

Encore aujourd’hui, on rencontre trop de personnes qui, lorsqu’on évoque les standards du Web, rétorquent : « *Et alors ? Je n’utilise pas tout ça et mon site marche ! Pourquoi diable devrais-je faire autrement ?* ».

Il s’agit là d’une vue bien étroite. Sans vouloir les vexer, cela revient à ne pas voir plus loin que le bout de son nez, à ne se préoccuper que de soi et de son environnement immédiat, ce qui est pour le moins inattendu s’agissant d’un contenu censé être accessible depuis le monde entier, souvent pour longtemps.

L’expression désormais consacrée « *HTML des années 1990* » est utilisée pour désigner ce mélange d’habitudes techniques aujourd’hui dépassées : balisage hétéroclite mêlant allègrement forme et fond, utilisant à mauvais escient certaines balises (ce qu’on appelle de la *soupe de balises*) ; emploi inapproprié ou incohérent de CSS ; surabondance d’éléments *div* ou d’attributs *class* superflus (syndromes baptisés *divitis* et *classitis*) ; déclinaisons manuelles ou à peine automatisées des pages suivant les navigateurs visés ; et bien d’autres usages, que je ne saurais citer tous ici.

Cette façon de faire, fruit d’une approche fondamentalement empirique du développement web et d’une évolution souvent organique des sites, sans cohérence préalable, était peut-être inévitable pour la première génération du Web. Après tout, la version initiale d’un projet regorge souvent d’horreurs qu’il faut ensuite éliminer. Mais il ne s’agit pas ici que d’esthétique. Les conséquences pénibles de cette approche sont nombreuses, et accablent aujourd’hui encore un grand nombre de projets et de sociétés qui persistent à ne pas évoluer :

- Faute d’une utilisation intelligente de CSS et de JavaScript, les pages sont beaucoup trop lourdes, constituées pour 10 % ou moins de contenu véritable. Impact : le coût de la bande passante pour votre site. Pour un site très visité (à partir du million de visiteurs uniques par mois), le superflu atteint un tel volume que son coût se chiffre fréquemment en dizaines voire en centaines de milliers d’euros par mois.
- Un balisage lourd ou rigide, ainsi qu’un emploi inadapté de CSS, créent des pages s’affichant de diverses façons en fonction du navigateur, sans parler des modes de consultation alternatifs répandus : assistant personnel (PDA), téléphone mobile 3G, borne Internet sans souris dans un espace public, *Tablet PC*, impression papier pour lecture ultérieure, et j’en oublie. Pour toucher une vaste audience, il faut alors en passer par la spécialisation de chaque page, travail ô combien fastidieux aux conséquences néfastes : d’une part il multiplie l’espace disque nécessaire, d’autre part il est généralement traité à la légère, de sorte qu’immanquablement certaines versions des pages seront de piètre qualité, voire pas à jour.

- Une mauvaise utilisation des CSS entraîne généralement une intrusion de l'aspect dans le contenu, et rend l'apparence des pages difficile à modifier globalement. Toute refonte de la charte graphique d'un site devient un cauchemar, à force de devoir dénicher tous les styles en ligne et les balises de mise en forme restées cachées au fond d'une page.
- À moins d'avoir été profondément sensibilisé à la question, une équipe de conception et développement de sites web aura tendance à enfreindre à tour de bras les règles d'or de l'accessibilité. Les pages seront donc difficilement exploitables par les non-voyants, les malvoyants, les personnes souffrant d'un handicap, même léger, rendant inenvisageable l'utilisation de la souris, mais aussi par les programmes de traitement automatique... Sachant que le plus grand internaute aveugle de la planète s'appelle Google, les sites peu accessibles sont par conséquent beaucoup moins bien classés dans les moteurs de recherche que s'ils respectaient les principes fondamentaux d'accessibilité. N'oublions pas que le Web est censé, par définition, être accessible par tous. Dans *World Wide Web*, il y a *World*.

À l'inverse, faire évoluer ses méthodes de travail pour garantir un niveau certain de qualité, pour trouver une façon plus moderne et finalement plus *facile* de réaliser des sites web, cela produit rapidement des bénéfices :

- Un site séparant clairement le contenu (balisage XHTML) de la forme (feuilles CSS) et du comportement (scripts JS, c'est-à-dire JavaScript) produira nécessairement des pages infiniment plus légères, sans parler de l'efficacité accrue des stratégies de cache des navigateurs devant un découpage des données en fichiers distincts, qui deviennent par ailleurs déportables sur des réseaux de distribution de contenu (ou CDN, *Content Delivery Networks*) tels que ceux de Yahoo!, Google ou Akamai, qui garantissent un téléchargement rapide et réduisent donc la charge de vos serveurs. Après avoir refondu complètement son site, ESPN, la principale chaîne de sports aux États-Unis, a augmenté son audience tout en divisant à tel point ses coûts de bande passante que l'économie mensuelle, malgré un tarif extrêmement avantageux, se chiffrait en dizaines de milliers de dollars ! (Pour les détails : <http://www.mikeindustries.com/blog/archive/2003/06/espn-interview>).
- Une utilisation appropriée des CSS implique qu'on a *une seule page XHTML*. J'insiste : *une seule*. Si vous croyez encore qu'il s'agit d'un mythe, allez donc faire un tour sur le CSS Zen Garden (<http://www.csszengarden.com>). Lorsque vous aurez essayé une petite vingtaine de thèmes, réalisez que la seule chose qui change, c'est la feuille de style. La page HTML est strictement la même. Il ne s'agit pas seulement d'offrir des thèmes, mais bien de proposer une vue adaptée de la page pour de nombreux usages et périphériques de consultation : page agréable à l'impression, mais aussi sur un petit écran (on pense aux PDA et aux téléphones mobiles), ou avec des modes spéciaux pour les mal-voyants (e.g. contraste fort, fond noir, etc.).

Puisqu'il n'y a qu'une seule page, elle est fatallement à jour, et les versions alternatives sont donc sur un pied d'égalité. Tout en gagnant de l'audience, vous la traitez mieux en garantissant le même contenu pour tous.

- Une prise en compte systématique de l'accessibilité, qui elle non plus n'entrave en rien la page, facilite la vie aux utilisateurs touchés par un handicap quelconque (plus de 20 % des internautes aux États-Unis et en France, selon certaines études). Couplée à l'emploi d'un balisage sémantique, elle signifie aussi que l'indexation de la page par les moteurs de recherche sera de bien meilleure qualité, augmentant votre visibilité et donc vos revenus potentiels.

Mercantilisme... aveugle ?

Le patron qui éructait, lors d'une conférence, « on vend des écrans plasma, on s'en fout des aveugles, ce ne sont pas nos clients ! » s'est vu gratifier d'une réponse cinglante : le mal-voyant voire non-voyant n'en est pas moins internaute, et s'il ne peut offrir à un proche un bel écran acheté sur votre site, il l'achètera ailleurs. La réflexion vaut, évidemment, pour tous les handicaps...

Et les navigateurs, qu'en pensent-t-ils ?

Disons qu'ils sont partagés. D'un côté, on trouve les navigateurs libres accompagnés de quelques navigateurs commerciaux traditionnellement respectueux des standards : Mozilla, Firefox, Camino, Konqueror, Opera et Safari, pour ne citer qu'eux. De l'autre, on trouve un navigateur commercial qui ne s'est réveillé que vers 2006, après une léthargie qui aura duré environ 7 ans, j'ai nommé MSIE.

La situation n'est toutefois pas si claire : tous les navigateurs n'ont pas le même niveau de prise en charge pour tous les standards, et le travail sur MSIE a repris à partir de la version 7, avec des progrès encore bien plus importants côté standards pour la version 8 à venir. Dressons ici un rapide portrait des principaux navigateurs au regard des standards. Gardez à l'esprit que cette situation évolue rapidement, et que vous aurez intérêt à suivre l'actualité des principaux navigateurs pour vous tenir à jour.

Notez que tous les navigateurs ci-après supportent XMLHttpRequest. MSIE utilise encore un ActiveX qui deviendra un objet natif JavaScript standard dans IE7, tandis que les autres navigateurs utilisent déjà un objet natif JavaScript.

On le voit, MSIE fait de gros progrès avec sa prochaine version 8, mais reste loin derrière les autres en termes de JavaScript et de DOM. Il ne faut pas s'étonner si Konqueror gagne du terrain chez les utilisateurs de Linux, et si Firefox continue, après 4 ans de vie, à grignoter des parts marché (27 % en France, 34 % en Europe soit près de 150 millions d'internautes, et jusqu'à 46 % dans certains pays). Qui-conque continue à développer un site Internet au mépris des standards ferait mieux de ne pas avoir trop d'ambitions commerciales...

Mozilla/Firefox/Camino

Ils utilisent peu ou prou le même moteur, même si la suite Mozilla a été abandonnée par la Fondation et que sa mise à jour est désormais assurée par une communauté de volontaires, qui peuvent parfois mettre du temps à intégrer les nouveautés de Firefox dans la suite complète (projet SeaMonkey). Quant à Camino, c'est un Firefox spécialisé Mac OS X, globalement équivalent côté standards. On parle ici de Firefox 3.

(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Excellent support du 2.1, et support de plusieurs modules 3.0.
JavaScript	Naturellement le meilleur, puisque le chef d'orchestre de la technologie travaille pour la Fondation Mozilla. Toujours à jour sur la dernière version. Firefox 3.0 prend en charge JS 1.8, restant ainsi le navigateur le plus avancé sur la question.
DOM	Très bon support du 2, support partiel du 3.

Safari 3.1

(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Très bon support du 2.1, hormis quelques aspects encore exotiques (notamment les styles audio).
JavaScript	Bon support du 1.5.
DOM	Très bon support du 2, support partiel du 3.

Opera 9.5

À noter qu'Opera propose une excellente page pour suivre sa compatibilité aux standards : <http://www.opera.com/docs/specs/>.

(X)HTML	Très bon support, à hauteur des versions récentes.
CSS	Excellent support de CSS 2.1.
JavaScript	Prend en charge ECMA 262 3 rd , soit JS 1.5.
DOM	Très bon support du niveau 2, bon support du 3.

Konqueror 4

(X)HTML	Très bon support, à hauteur des versions récentes
CSS	Excellent support de CSS 2.1, support partiel de CSS 3.
JavaScript	Bon support du 1.5.
DOM	Très bon support du 2, support partiel du 3.

Internet Explorer 6

(X)HTML	Très bon support, à hauteur des versions récentes. Un souci avec les prologues XML, sans grande importance.
CSS	Support CSS 1, très partiel de CSS 2.
JavaScript	Jscript 6.0, pas totalement compatible JavaScript, fonctionnellement entre JS 1.3 et 1.5.
DOM	Support correct du niveau 2, mais persiste à se comporter parfois différemment du standard !

Internet Explorer 7

(X)HTML	Plus de souci de prologues
CSS	Support quasi total de CSS 1, et assez bon de CSS 2.1.
JavaScript et DOM	Aucune amélioration significative depuis la version 6. Entre le DOM des objets <code>select</code> , le <code>getElementById</code> qui utilise aussi l'attribut <code>name</code> , les prototypes non modifiables des objets natifs, le modèle événementiel propriétaire (notamment pas de <code>addEventListener</code>) ou l'absence très pénible de DOM niveau 3 XPath, il y a de quoi faire pour IE8...

Internet Explorer 8 (sur base de la beta)

(X)HTML	beaux progrès, avec entre autres une implémentation partielle de HTML 5.
CSS	très gros progrès sur CSS 2.1 (passe Acid2), et exploite le mode « conforme aux standards » <i>par défaut</i> .
JavaScript et DOM	progrès sur quelques bogues incontournables du DOM et sur le ramasse-miettes, mais rien pour le moment quant à l'API d'événements, ni sur le langage JavaScript lui-même... De vrais outils de débogage cependant, largement inspirés de Firebug !

Quelques mots sur les dernières versions

Voici un rapide tour d'horizon des versions en cours et à venir pour les principaux standards. Là encore, un peu de veille sera votre meilleur atout.

- **HTML est en version 4.01** (décembre 1999), **et la version 5 est en plein chantier**. La prise en charge de cette future version par les navigateurs est déjà prometteuse (que ce soit dans Firefox 3, Opera 9.5, Safari 3.1 ou Internet Explorer 8).
- **XHTML est en version 1.1**. La plupart des navigateurs implémentent au moins la 1.0. La 1.1 est plus stricte et demande normalement un type MIME distinct, qui panique notamment MSIE 6 ! En revanche, certains aspects de la prochaine version de XHTML, la 2.0, sont dénigrés par le plus grand nombre, au motif principal qu'elles cassent vraiment trop la compatibilité descendante sans grand avantage en retour. La spécification est d'ailleurs plus ou moins gelée depuis 2 ans.
- **CSS est en version 2.1**, avec beaucoup de travail autour des **37 modules composant CSS 3.0**. Le calendrier de sortie de ces modules à titre de recommandations s'étalera probablement sur au moins 5 ans... Les dernières versions de tous les navigateurs sont « au taquet » sur CSS, avec évidemment un retard certain pour MSIE 8, retard cependant bien moins critique que par le passé.
- **DOM est au niveau 2, et avance bien au niveau 3**, plusieurs modules étant terminés, dont le *Core*. La plupart des navigateurs s'attaquent fermement à ce nouveau niveau, et même MSIE devrait rattraper un peu son retard prochainement.
- **JavaScript est en version 1.8**, actuellement uniquement pris en charge par Firefox 3.0, mais la disponibilité de bibliothèques Java et C++ toutes prêtes (par ex.

Rhino) facilitent l'intégration par d'autres navigateurs. Les versions 1.7 et 1.8 apportent quelques grandes nouveautés, mais sont encore loin de la 2.0 (dont le nom officiel est « ECMAScript 4e Edition », ou « ES4 » pour faire court), dont la sortie ne cesse d'être reportée, mais qui sera impressionnante !

Qu'est-ce que le « Web 2.0 » ?

Le terme « Web 2.0 », qui a envahi la presse et les sites spécialisés, décrit en réalité deux phénomènes distincts.

D'une part il y a cette évolution profonde des interfaces utilisateur proposées en ligne, qui rattrapent en convivialité et en interactivité celles qu'on trouve sur des applications plus classiques (applications dites « desktop », au sens où elles s'exécutent en local sur la machine de l'utilisateur), ou même sur celles qui équipent des périphériques légers (téléphones mobiles, assistants personnels, etc).

Glisser-déplacer, complétion automatique, création dynamique d'images, personnalisation à la volée de l'interface, exécutions en parallèle : autant de comportements que nous avons pris l'habitude de trouver dans les applications, et qui manquaient cruellement — jusqu'à récemment — aux navigateurs. Ceux-ci étaient réduits à des rôles subalternes, à un sous-ensemble ridiculement étiqueté de possibilités bien définies. Et pourtant, les navigateurs ne sont pas plus bêtes que les autres programmes : nous les avons simplement sous-exploités jusqu'ici.

Cette première facette s'est récemment trouvé le nom de « RIA », pour *Rich Internet Application*, un terme initialement apparu chez Macromedia, depuis racheté par Adobe. Le navigateur devient véritablement une plate-forme applicative, comme le montrent des applicatifs comme Google Docs ou GMail. On trouve même désormais d'impressionnantes séries d'outils graphiques avancés (dessin vectoriel, illustration, retouche photo, montage vidéo...) entièrement en ligne. Certains mêmes n'hésitent plus à dire que le système d'exploitation (par exemple Windows®) n'est qu'un « ensemble de pilotes destiné à faire fonctionner le navigateur » !

L'autre facette, qui est la caractéristique principale du Web 2.0, est ce qu'on pourrait appeler *le Web aux mains des internautes*. On est passé d'un contenu créé par des « experts », à un contenu directement livré par l'internaute. Ainsi il n'y a pas si longtemps, consulter une page web constituait une expérience similaire à la lecture d'une page imprimée dans un magazine : on n'avait pas son mot à dire sur l'aspect. Cette barre de navigation sur la droite vous gâche la vue ? Ce bandeau de publicité vous énerve ? Le texte est trop petit, ou le contraste trop faible pour votre vue ? Tant pis pour vous ! Le concepteur graphique du site l'a voulu ainsi, et sa volonté fait loi. En fait, consulter une page web était encore *pire* que lire un magazine : sur ce dernier, au

moins bénéficiait-on de l' excellente résolution de l'impression, de sorte que les textes en petite casse restaient lisibles. Bien sûr, la plupart des navigateurs permettent de désactiver CSS ou d'utiliser une feuille de style personnelle, ou encore de zoomer le texte, voire toute la page (images comprises), mais c'est une piètre consolation.

Et voilà que de nouveaux usages apparaissent, qui donnent enfin à l'internaute la haute main sur l'aspect final de la page *sur son navigateur*. *Exit*, les parties superflues et irritantes ! Agrandi, le texte principal écrit trop petit ! Et tant qu'à faire, augmentons la marge entre les paragraphes et aérons le texte en changeant l'interligne ! À l'aide d'outils dédiés, par exemple les extensions GreaseMonkey (<http://greasemonkey.mozdev.org/>) et Platypus (<http://platypus.mozdev.org/>) pour Firefox, le visiteur peut ajuster comme bon lui semble l'aspect d'une page, et rendre ces ajustements automatiques en prévision de ses visites ultérieures.

Par ailleurs, les internautes peuvent maintenant contribuer à l'actualité du Web, tant grâce à la facilité de publication qu'offrent des outils comme les blogs, qu'au travers d'annuaires de pages très dynamiques basés sur des votes de popularité (par exemple Digg ou Reddit). Le principe est simple : ces annuaires permettent à tout un chacun de « voter » pour une page quelconque du Web, afin de dire quelque chose comme « Hé ! J'ai aimé cette page ! ». Les annuaires maintiennent alors une liste, par popularité décroissante, des pages ainsi signalées.

Si un nombre massif d'internautes votent pour une même page, celle-ci apparaît fatalement en excellente position dans l'annuaire qui a recueilli les votes. Le résultat net est séduisant : les hauts de liste pour ces annuaires sont là-haut parce que leur contenu a intéressé, amusé ou marqué un maximum de gens. Statistiquement, il a donc toutes les chances de vous intéresser, vous aussi.

Sous un angle plus politique, cela signifie que les gros titres ne sont plus confiés à un comité de rédaction, si facile à instrumentaliser. Pour acquérir une telle visibilité, fût-elle éphémère, la page n'a d'autre choix que de plaire à beaucoup de monde. C'est un système très démocratique.

Les principaux sites de ce type : del.icio.us (<http://del.icio.us/>) et Digg (<http://www.digg.com/>, plus orienté technologie), pour n'en citer que deux, sont déjà extrêmement visités (plusieurs dizaines de millions de visiteurs uniques par jour). Du coup, de nombreux blogs, magazines en ligne et autres sites au contenu très dynamique affichent systématiquement sur leurs pages des liens graphiques aisément reconnaissables pour faciliter (et donc encourager) le vote de l'internaute auprès des principaux annuaires.

Les sites Technorati et del.icio.us figurent également parmi les pionniers d'un nouvel usage qui se répand rapidement : le *tagging*. Il s'agit de permettre aux internautes de qualifier une page à coup de mots-clés, pour obtenir un système riche de références croisées et de catégories tous azimuts, bien plus souple que les hiérarchies de catégories

habituelles. Tous les outils de blog, comme Typo (<http://typosphere.org/>) ou Dotclear 2 (<http://www.dotclear.net/>), proposent désormais l'affectation de *tags* (étiquettes) aux billets.

Le Web 2.0, tout comme Firefox à sa sortie, vous invite finalement à « reprendre la main sur le Web ! »

Vue d'ensemble, chapitre par chapitre

Pour finir cet avant-propos (un peu dodu, je vous l'accorde), je vous propose de jeter un coup d'œil général à la structure de l'ouvrage, en soulignant son articulation et le rôle de chaque chapitre.

- **Le chapitre 1, « Pourquoi et comment relever le défi du Web 2.0 ? »** pose la problématique et les enjeux. Il s'agit de bien saisir le saut conceptuel entre les sites classiques et le Web 2.0 ; après de nombreux exemples illustrés et le positionnement des principales technologies dans l'architecture globale d'un développement web, le chapitre démystifie Ajax et termine en dressant un plan d'action, autour de cet ouvrage, pour vous aider à tirer le maximum de votre lecture.

Première partie : donner vie aux pages

Quatre chapitres visent à s'assurer que vous maîtrisez bien les piliers désormais classiques sur lesquels se construit aujourd'hui Ajax. À moins que vous ne soyez véritablement un expert en JavaScript et DOM, parfaitement respectueux des standards qui les gouvernent, je ne saurais trop vous recommander de ne *pas* faire l'impasse sur ces chapitres, au seul prétexte que vous croyez les connaître. Il y a fort à parier que vous allez y apprendre quelque chose.

- **Le chapitre 2, « Ne prenez pas JavaScript pour ce qu'il n'est pas »** présente en détail ce langage si mal connu, accablé *d'a priori* et souvent bien mal employé. Ce chapitre est très riche en conseils et astuces méthodologiques, et prend soin de vous aider à réaliser une couche « comportement » la plus propre et la plus élégante possible.
- **Le chapitre 3, « Manipuler dynamiquement la page avec le DOM »,** nous ouvre les voies royales qui mènent aux pages véritablement dynamiques, dont le contenu évolue rapidement, entièrement côté client. De nombreux exemples pour des besoins concrets sont présentés. Des conseils précieux et un point sur les problèmes résiduels de compatibilité terminent ce chapitre.
- **Le chapitre 4** présente la quasi totalité de **Prototype**, sans doute la plus utile des bibliothèques JavaScript les plus répandues. Grâce à elle, nous allons apprendre à

réaliser du code JavaScript, non seulement plus portable que nos précédents exemples, mais encore plus élégant, plus concis, plus expressif, et d'une façon générale tellement plus agréable à écrire...

- **Le chapitre 5** tente de combler une terrible lacune de la plupart des ressources actuelles, en vous montrant dans le détail comment **déboguer votre JavaScript**, que ce soit sur Firefox, Safari, Opera ou Internet Explorer. Comme dans le reste de l'ouvrage, plus qu'un simple passage en revue des outils, c'est une véritable méthodologie de travail qui sera proposée.

Deuxième partie : Ajax, ou l'art de chuchoter discrètement

Une fois vos bases techniques bien solides et confortables, vous allez pouvoir vous plonger dans ce qui constitue, pour beaucoup, la partie la plus visible d'Ajax : les requêtes asynchrones en arrière-plan. C'est grâce à elles que nos pages semblent enfin capables de « faire plusieurs choses en même temps », et n'ont plus autant besoin de se recharger intégralement.

- **Le chapitre 6, « Les mains dans le cambouis avec XMLHttpRequest »**, vous emmène jusqu'aux tréfonds de la technologie responsable des requêtes asynchrones. C'est l'occasion de découvrir une autre technologie de pointe, très agréable elle aussi : le langage Ruby, dont nous nous servirons pour créer, avec une déconcertante facilité, un serveur web à contenus dynamiques pour nos tests.
- **Le chapitre 7, « Ajax tout en souplesse avec Prototype »**, nous fait passer à la vitesse supérieure ! Puisque nous maîtrisons désormais les rouages, nous allons pouvoir délaisser le cambouis pour faire des bonds spectaculaires en productivité avec les facilités Ajax de Prototype.
- **Le chapitre 8, « Une ergonomie de rêve avec script.aculo.us »** explore l'incroyable bibliothèque d'effets visuels et de comportements avancés proposée par script.aculo.us. Ce chapitre vous emmène par ailleurs plus loin dans la réflexion, autour des usages pertinents ou malvenus d'Ajax et des limites de son utilisation.

Troisième partie : Parler au reste du monde

C'est un peu la partie bonus, qui va au-delà de la technologie Ajax pour explorer des usages concrets et de plus en plus fréquents. L'idée, c'est que nos pages n'ont aucune raison de se limiter à notre serveur, et peuvent discuter tout aussi aisément avec n'importe quel site et n'importe quel service prévu à cet effet.

- **Le chapitre 9, « WebServices et REST : nous ne sommes plus seuls »**, illustre cette idée en présentant ces deux technologies pour s'atteler ensuite à faire profiter nos pages des possibilités de recherche d'Amazon, de prévision de The Weather Channel, et des bibliothèques d'images de Flickr.

- **Le chapitre 10, « L'information à la carte : flux RSS et Atom »,** présente les deux principaux formats de flux pour mettre en œuvre une syndication de contenus (blogs et autres) directement sur nos pages.
- **Le chapitre 11, « Mashups et API 100 % JavaScript »,** illustre la nouvelle tendance à la réutilisation de services externes directement côté client, au travers du célèbre service Google Maps et du plus récent (mais tout aussi sympathique) Google Charts.

Des annexes pour le débutant comme pour l'expert

Sur cinq annexes, deux visent à aider le lecteur auquel manqueraient quelques bases, tandis que les trois dernières donnent à tous des compétences recherchées.

- **L'annexe A, « Bien baliser votre contenu : XHTML sémantique »,** redonne les bases du XHTML et insiste lourdement sur l'importance d'un balisage non seulement valide, mais surtout sémantique. Après avoir succinctement donné la liste des balises pour mémoire, elle fournit également quelques cas concrets de balisage impeccable, correspondant à des besoins récurrents.
- **L'annexe B, « Un aspect irréprochable et flexible : CSS 2.1 »,** joue le même rôle vis-à-vis de CSS, et donc de la mise en forme. Le vocabulaire est précisé, avant d'attaquer suffisamment les fondamentaux : structure des règles, principe de cascade et modèle de boîtes. Une liste concise des sélecteurs et propriétés permet de ne pas trop patauger dans les exemples du reste de l'ouvrage.
- **L'annexe C, « Le plus de l'expert : savoir lire une spécification »,** apporte une réelle plus-value en vous apprenant à lire les principaux formats de spécification pour les standards du Web et à naviguer au sein de ces documents parfois complexes, qui font souvent appel à des syntaxes particulières. Être à l'aise avec ces documents présente de nombreux avantages, et constitue *une compétence encore trop rare*.
- **L'annexe D, « Développer avec son navigateur web »,** fait le point sur les possibilités plus ou moins riches pour votre productivité de développeur web sur les principaux navigateurs : gestion du cache, extensions, outils complémentaires de débogage et de test, autant d'outils qui sont passés en revue. *À lire impérativement, en fait, avant de démarrer le livre !*
- **L'annexe E enfin, « Tour d'horizon des autres frameworks JavaScript »,** donne un aperçu des frameworks les plus populaires après Prototype et script.aculo.us, avec notamment jQuery, YUI, Ext JS et Dōjō.

J'ai fait de mon mieux pour que vous retrouviez dans cet ouvrage autant d'informations techniques, concrètes et de qualité, que ce que je fournissais à mes étudiants dans mes cours.

Aller plus loin...

On ne le répétera jamais assez, tout l'ouvrage tente de vous insuffler le constant souci de la qualité, de l'élégance, de l'efficacité, au travers de nombreux conseils méthodologiques et choix techniques savamment orientés. L'objectif n'est rien moins que vous rendre meilleur(e) que vos concurrents !

Dans le même esprit, la plupart des chapitres se terminent par une section « Pour aller plus loin... », qui donne la liste des ouvrages et ressources en ligne permettant d'approfondir les sujets explorés.

À propos des exemples de code

L'ensemble des codes source de ce livre est disponible dans une archive mise en place sur le site web des Éditions Eyrolles, accessible depuis la page de l'ouvrage. Certains chapitres n'utilisent que de courts extraits (par exemple, le chapitre 4 sur Prototype), mais l'archive fournit toujours des pages de test complètes.

Ces exemples ont tous été testés sur Firefox 2, Firefox 3, Safari 3.1, MSIE 6, MSIE 7, MSIE 8b1 Opera 9.5 et Konqueror 3.5.2. Lorsque certaines contraintes sont incontournables, leur impact est précisé dans le texte du livre.

Par ailleurs, les bibliothèques Prototype et script.aculo.us qui y figurent sont parfois plus récentes que leur dernière version stable publique (1.6.0.3 et 1.8.1 respectivement). L'archive de codes source vous fournit aussi ces versions à part, dans un répertoire `bibliothèques_seules` à la racine de l'archive.

Remerciements

Ce livre n'aurait pas vu le jour sans la confiance que m'ont témoignée Muriel Shan Sei Fan et Éric Sulpice. Leur bonne humeur, leur amour de l'informatique et leur dynamisme m'ont d'abord donné envie d'écrire pour Eyrolles, et par la suite grandement facilité la tâche. Un gros merci à Muriel, notamment pour avoir fait sentir très tôt le besoin d'un *extreme makeover* sur la table des matières !

Xavier Borderie et Richard Piacentini ont eu la gentillesse d'assurer la relecture technique de la première édition. Raphaël Goetter, gourou des CSS, a également accepté de relire l'annexe B, en dépit de son planning de ministre. Le livre a énormément bénéficié de leurs apports et remarques constructives. Ce que vous y aimerez, vous le leur devrez certainement. Si certaines parties vous déçoivent, la faute sera mienne. J'adresse également toute ma gratitude à Tristan Nitot pour avoir accepté de rédiger la préface, ce que je considère comme un bel honneur.

Enfin, ma compagne Élodie Jaubert a supporté mon manque de disponibilité pendant les quelque trois mois d'écriture, et m'a soutenu sans faillir avec beaucoup d'amour, au point même d'accepter, au cœur de la tourmente, de devenir ma femme. Ce livre est là, avant tout, grâce à elle.

Et pour cette deuxième édition...

Déjà deux ans... Et 5 000 exemplaires ! Le livre s'est établi une solide petite réputation de référence et les commentaires en ligne et courriels que j'ai reçus vont au-delà de mes espérances.

C'est donc avant tout les lecteurs de la première édition que je souhaite remercier ici, car sans eux, on ne remettrait pas le couvert. J'aime à croire que cette nouvelle mouture a suffisamment de bonnes choses en elle pour se justifier pleinement à leurs yeux ! Élodie, elle, croit toujours en moi et me donne la force d'avancer, d'écrire, de partager. Merci pour tout, mon ange.

1

Pourquoi et comment relever le défi du Web 2.0 ?

Le Web 2.0, c'est bien, mais quels en sont les problématiques et les enjeux ? Que peut-on attendre à présent des sites, quelles technologies doit-on maîtriser (et peut-être apprendre à nouveau, apprendre mieux), et comment interopèrent-elles, notamment dans le cadre d'Ajax ? Ce chapitre tente de répondre à toutes ces questions, et termine en établissant un plan d'action simple, s'appuyant sur cet ouvrage, pour vous aider à devenir un véritable expert des technologies Web 2.0.

Avant/après : quelques scénarios frappants

Afin de bien fixer les idées, explorons ensemble quelques services faisant un emploi efficace (et plutôt emblématique) des possibilités d'Ajax.

La saisie assistée : complétion automatique de texte

Une des principales utilisations novatrices d'Ajax est la saisie assistée, également appelée « complétion automatique de texte ». Le principe est simple, et courant dans les applications classiques : au fur et à mesure de la frappe, une série de valeurs finales possibles est proposée à l'utilisateur, qui correspond à ce qu'il ou elle a tapé jusqu'ici.

C'est le comportement qu'on attend d'un simple téléphone mobile (avec le fameux mode T9 pour la saisie des messages) tandis qu'il restait dramatiquement absent des pages web, lesquelles évoluent pourtant dans un environnement bien plus sophistiqué.

Voyons un premier exemple, avec www.ratp.info, le site de la RATP, la régie des transports en Île-de-France. Leur moteur de recherche d'itinéraires permet de préciser des adresses complètes, des stations ou des lieux comme points de départ et d'arrivée. En mode station par exemple, au fil de la frappe, une liste de possibilités est affichée qui permet de saisir rapidement la station visée.

Imaginons que l'on souhaite partir de la station « Arts et Métiers » à Paris. À peine a-t-on tapé ar que la liste (qui s'affiche par-dessus la saisie, ce qui est un choix discutable) présente l'aspect suivant :

Figure 1–1

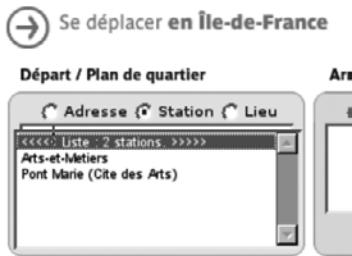
Une saisie assistée sur le site de la RATP, après frappe de « ar »



Ajoutons simplement le t, et la liste devient :

Figure 1–2

La saisie assistée après frappe de « art »



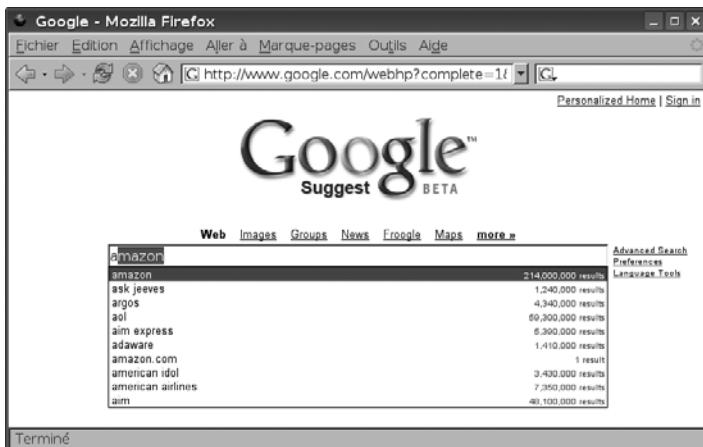
Il ne nous reste plus qu'à sélectionner directement la station (ce qui peut se faire en deux touches de clavier ou d'un clic de souris), opération bien entendu plus rapide qu'une saisie complète et qui réduit notamment le risque d'erreurs de frappe, améliorant ainsi la pertinence du moteur de recherche d'itinéraires.

Un exemple plus connu, et peut-être plus impressionnant, est Google Suggest, fonction proposée par les laboratoires de Google (<http://labs.google.com>, qui propose une foule de fonctions avancées encore en rodage).

Google Suggest est un mode spécial d'utilisation de la version anglophone du moteur de recherche : <http://www.google.com/webhp?complete=1&hl=en>. Voici ce qui se passe (instantanément ou presque !) lorsqu'on démarre une recherche sur Ajax en tapant le a initial :

Figure 1–3

Saisie assistée avec Google Suggest, après frappe de « a »

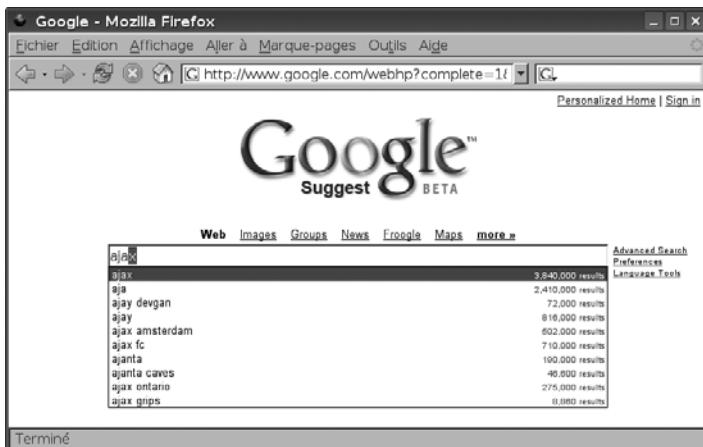


La liste de propositions, qui est longue, inclut également le nombre de résultats, ce qui permet de rendre moins ambiguë la recherche (par exemple, en cas de doute dans l'orthographe de Johannesburg, la capitale de l'Afrique du Sud, à peine a-t-on tapé *johan* qu'on voit l'orthographe correcte obtenir près de 7 millions de résultats, tandis qu'en ajoutant immédiatement un *e*, on tombe sur des recherches ne dépassant pas les dizaines de milliers).

Voici l'affichage de Google Suggest alors que nous continuons notre recherche sur Ajax, en ayant tapé *aja* :

Figure 1–4

Saisie assistée avec Google Suggest, après frappe de « aja »



Notez que sur une connexion aujourd’hui classique voire minimale (ADSL d’au moins 512 kbit/s), les requêtes en arrière-plan effectuées par le moteur de suggestion n’entraînent pas le moins du monde le confort de frappe.

Depuis sa version 2, Firefox utilise ce même comportement dans sa barre de recherche Google (et d’autres navigateurs ont suivi).

Le chargement à la volée

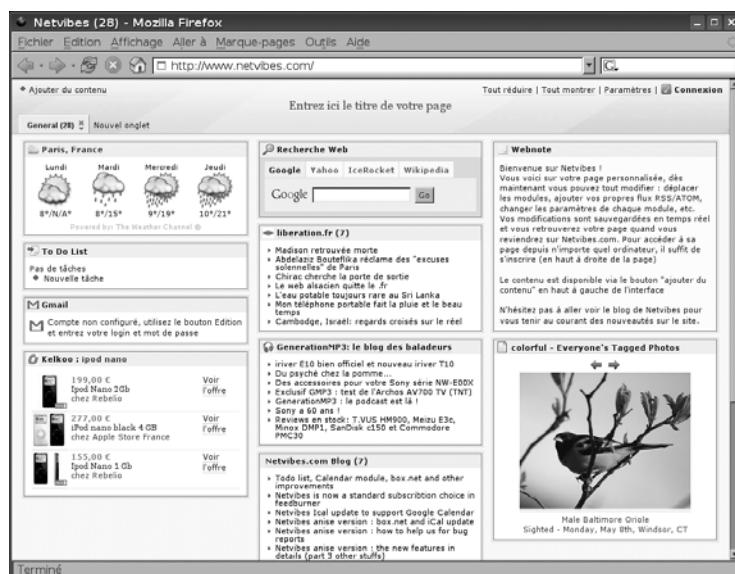
Une autre utilisation très courante d’Ajax réside dans le chargement de données à la volée. On peut d’ailleurs considérer la saisie assistée comme un cas particulier de chargement à la volée. D’une façon générale, l’idée reste l’obtention de contenu suite à une action utilisateur sans exiger le rechargeement complet de la page.

Il peut s’agir de n’importe quel contenu : liste de propositions (comme nous l’avons vu précédemment), informations dynamiques (données météo, valeurs boursières), articles issus de flux RSS ou Atom (blogs, modifications apportées à un référentiel de sources, journaux d’information en ligne), graphiques (cartes, prises de vue par satellite, niveaux de jeu en ligne)... La seule limite reste l’imagination (et, dans une mesure chaque jour plus négligeable, la bande passante) !

Deux exemples incontournables donnent un aperçu des possibilités.

Tout d’abord Netvibes, jeune société française spécialisée dans les technologies Web 2.0, dont la page d’accueil fournit à tout un chacun un portail personnel en ligne entièrement personnalisable, tant dans le contenu que pour la disposition visuelle de ce contenu. Voici la page d’accueil par défaut :

Figure 1–5
La page d’accueil
par défaut de Netvibes



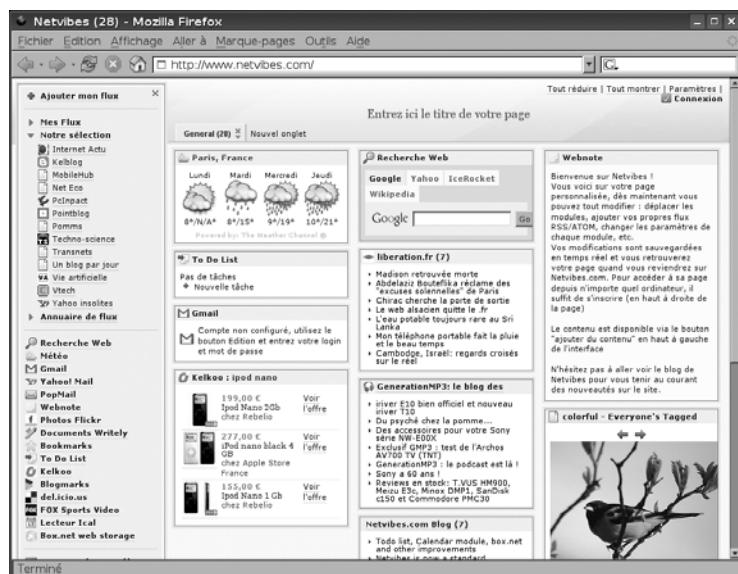
Notez la structure de la page :

- Une série de pavés, qui peuvent être déplacés librement et à volonté, par simple glisser-déplacer, afin d'arranger la disposition du contenu comme bon nous semble.
- Un titre personnalisable (il suffit de cliquer dessus pour le rendre éditable immédiatement).
- Des liens dans le haut permettant d'ajouter du contenu, de minimiser/restaurer les différents pavés, de régler quelques paramètres généraux (tout ceci sans recharger la page d'ensemble) et de se connecter à son compte Netvibes.
- Des onglets pour catégoriser le contenu afin d'éviter une page trop lourde visuellement.
- Pas de bouton de sauvegarde.

Quant aux contenus disponibles, ils sont de natures très diverses : météo, liste de choses à faire, consultation de courriel (*via* Gmail), comparateurs de prix, dictionnaires et encyclopédies, blogs, notes et même des catalogues de photos (*via* Flickr) ! Voici d'ailleurs l'interface d'ajout de contenu, qui apparaît à gauche de la page lorsqu'on active le lien correspondant :

Figure 1–6

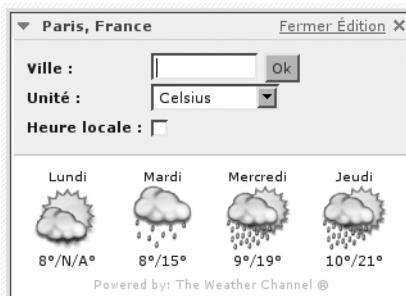
L'interface d'ajout de contenu de Netvibes



Chaque pavé dispose de propriétés spécifiques pour ajuster son comportement et son affichage, comme en témoignent les figures suivantes.

Figure 1–7

Modification des propriétés d'un pavé météo

**Figure 1–8**

Modification des propriétés d'un pavé de tâches à faire

**Figure 1–9**

Modification des propriétés d'un pavé de recherche des meilleurs prix

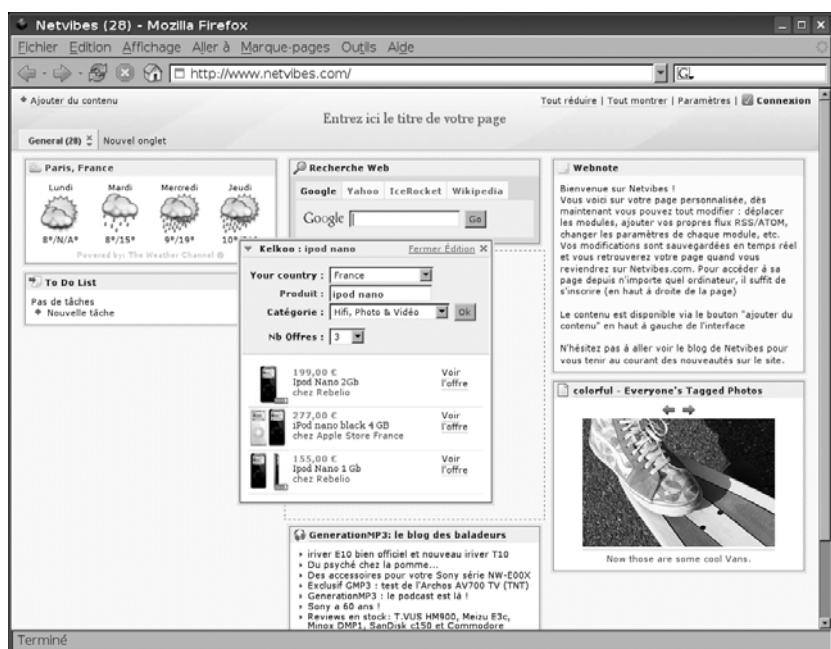


Chaque pavé peut être déplacé à l'aide d'un simple glisser-déplacer. Après avoir retiré quelques pavés, la figure 1–10 montre ce que l'on obtient tandis que l'on en déplace un.

Notez la délimitation en pointillés de l'emplacement cible, qui n'est peut-être pas très bien rendue à l'impression de cet ouvrage. Qu'à cela ne tienne : allez sur le site – qui ne nécessite aucune inscription – et essayez vous-même !

Figure 1–10

Déplacement d'un pavé par glisser-déplacer



Ces images permettent difficilement de rendre compte de l'impression que fait l'utilisation d'un service comme Netvibes. Pour dire les choses simplement, on a l'impression d'utiliser une application normale. En d'autres termes, la page ne souffre pas des limitations que nous associons habituellement au contenu web. Pas de rechargement global, très peu de délais, beaucoup de réactivité, une large place attribuée aux manipulations à la souris : ce ne sont là que quelques aspects qui, pour évidents qu'ils soient dans nos applications favorites, nous surprennent encore dans une page affichée par un navigateur. La seule zone d'ombre du site, qui tient dans une certaine tendance à la *divitis*¹, n'affecte en rien le confort de manipulation proposé à l'utilisateur.

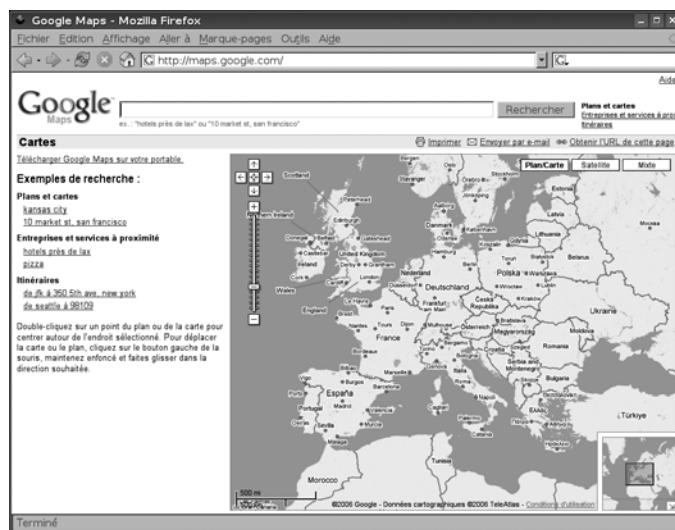
Un autre exemple phare du chargement de contenu à la volée grâce à Ajax est le moteur de cartographie interactive Google Maps. Ce service en ligne permet d'effectuer des recherches géographiques (comme « hôtels à Paris » ou « 61 bld Saint-Germain 75005 Paris ») et de les associer à une cartographie détaillée, qui peut même être mélangée à des prises de vue par satellite. Il est également possible de demander des itinéraires.

Voici la vue initiale de Google Maps, recentrée sur l'Europe (figure 1–11).

1. <http://fr.wikipedia.org/wiki/Divitis>

Figure 1-11

Google Maps, vue centrée sur l'Europe, faible niveau de zoom



Par glisser-déplacer, on peut se déplacer dans la carte, aussi loin que l'on souhaite. On peut également faire glisser le curseur sur la gauche pour augmenter le niveau de zoom. Tout ceci suppose bien sûr que le contenu affiché est récupéré au fur et à mesure, sans quoi le volume de données à obtenir serait tellement énorme que toute tentative serait vouée à l'échec.

Le type de carte évolue suivant le niveau de zoom. Ainsi, en se concentrant sur Paris, on obtient une vue plus « routière » :

Figure 1-12

Google Maps, vue centrée sur Paris, niveau de zoom moyen



Et en cherchant une adresse (ou un itinéraire), on obtient une vue détaillée, avec en prime un ballon (notez l'ombre portée : on fait décidément dans la finesse pour l'interface utilisateur !) situant précisément l'emplacement concerné :

Figure 1–13

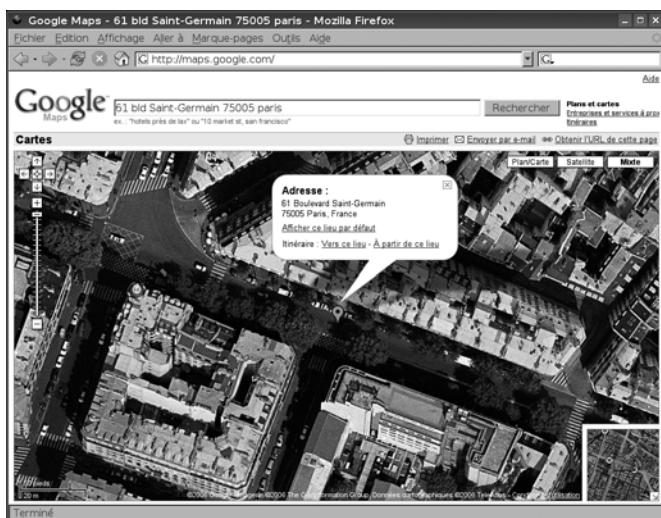
Google Maps, vue calée sur une adresse précise, zoom élevé



Google Maps permet également la reprise, et le mélange, des prises de vue par satellite qui font la base de Google Earth. Ainsi, pour ceux qui préfèrent du visuel à des cartes, ou pour ceux qui s'interrogent sur l'aspect de leur quartier, on peut passer en vue satellite ou mixte, et ce jusqu'à un niveau de détail impressionnant (1 pixel pour 50 cm) :

Figure 1–14

Google Maps, vue calée sur une adresse précise, en mode mixte, zoom maximal



Bienvenue sur le toit des éditions Eyrolles.

Le plaisir et le confort d'utilisation de ce service reposent sur l'interactivité forte qu'il permet : on utilise exclusivement la souris, on glisse, on zoomé, on se déplace. La page ne se recharge jamais dans son intégralité, pas d'attente intermédiaire, pas de page blanche. Sans des technologies comme Ajax, un tel service serait un enfer ergonomique.

La sauvegarde automatique

Un aspect important de nombreuses interfaces web reposant sur Ajax est la sauvegarde automatique. En d'autres termes, on trouve de moins en moins de boutons Valider, Envoyer, Sauvegarder ou Enregistrer. Toute saisie est transmise automatiquement au serveur, en arrière-plan, une fois qu'elle est considérée terminée.

Si vous vous promenez à travers l'interface de Netvibes, par exemple, vous verrez qu'il n'y a pas de bouton dont le rôle est d'envoyer vos modifications au serveur. Et ce pour une raison bien simple : ces envois ont lieu de toute façon, en arrière-plan. C'est la base d'Ajax. Vous voulez changer le titre de la page ? Cliquez dessus, tapez le nouveau titre, validez avec la touche Entrée ou cliquez simplement ailleurs pour annuler la sélection du titre, et c'est tout.

On retrouve un schéma similaire dans un nombre grandissant de services d'achat en ligne, en particulier pour les hypermarchés sur le Web. Faire des courses est un processus plus exigeant, ergonomiquement, que les achats en ligne classiques. Dans ces derniers, on achète un nombre assez restreint d'éléments distincts : qu'il s'agisse de livres, CD-Rom, DVD, matériels de sport ou billets pour des spectacles, le panier reste relativement léger. Tandis que pour des courses, il enfle rapidement, pour atteindre plusieurs dizaines d'éléments distincts.

Devoir subir un aller-retour global pour chaque ajout découragerait l'internaute, et les hypermarchés l'ont bien compris. Toutefois, ils ont généralement recours à la technique éprouvée (et littéralement d'un autre siècle) des cadres (*frames*) afin d'aboutir à ce résultat : un cadre est dédié au panier, tandis qu'un ou plusieurs autres cadres affichent les rayons et produits.

Ce type d'architecture est certes bien connu et maîtrisé des professionnels du Web, mais si les principaux avocats des standards du Web (W3C, WHAT WG, WaSP) et de l'accessibilité dénigrent les cadres, ce n'est pas sans raison.

Outre les obstacles majeurs à l'accessibilité qu'ils présentent (ils complexifient grandement la navigation, en particulier au clavier, ainsi que pour les utilisateurs malvoyants et les logiciels qui les assistent), les cadres ainsi utilisés engendrent une complexité importante dans le code JavaScript nécessaire, sans parler des problèmes de compatibilité entre navigateurs.

De plus en plus de boutiques en ligne font le saut vers une architecture tout Ajax qui permet de simplifier radicalement le code côté client, tant au niveau JavaScript que HTML, en particulier en utilisant des bibliothèques telles que Prototype ou des frameworks dédiés. Notons toutefois qu'un tel virage technologique doit être accompagné d'une politique efficace d'accessibilité, comme nous le verrons au chapitre 8.

Bien maîtriser ses outils clefs : XHTML, CSS, JS, DOM et Ajax

Réaliser un site Web 2.0 ne repose pas sur une seule technologie. Comme on le verra au chapitre 6, Ajax lui-même n'est que la combinaison intelligente et novatrice de composantes techniques qui ne datent pourtant pas toutes d'hier.

C'est précisément cet existant qui peut jouer des tours au développeur, car il peut donner l'illusion d'une maîtrise déjà acquise, et l'amener à sauter, ou tout au moins à survoler, les chapitres et annexes dédiés à XHTML, CSS, JavaScript et DOM. Or, traiter ces sujets à la hussarde, en se disant « ça, je connais déjà », constituerait à mon humble avis une erreur.

Permettez-moi ici un bref aparté. De 2002 à 2007, j'ai eu le plaisir d'enseigner dans une école d'ingénieurs en informatique en alternance située à Paris. En charge de la spécialisation SIGL (systèmes d'information et génie logiciel), j'accordais une attention particulière aux retours d'expérience de nos étudiants (environ 700 depuis mon arrivée) sur leurs stages (3 jours par semaine en entreprise sur toute la durée de l'année scolaire, soit 10 mois), notamment lorsque ces stages impliquaient du développement.

S'il est une impression qu'on retrouve très fréquemment dans leurs récits, c'est celle d'un faible niveau de compétences élémentaires en technologies web côté client (toutes celles dont traite cet ouvrage) chez leurs collègues, tant stagiaires que professionnels chevronnés. Mais si on leur demande d'étayer leur sentiment, on retrouve toujours les mêmes remarques : « il ne connaît rien au DOM », « elle n'a jamais entendu parler de balisage sémantique », « ils mettent des div partout, dont 95 % sont superflus », « ça ne marche que sur IE 6, et encore », « personne n'avait jamais ouvert une spécification du W3C, c'était totalement empirique », « à force de faire du Dreamweaver, ils étaient piégés quand ça ne marchait plus », etc.

Le constat fait réfléchir : il n'est pas rare de voir étudiants, jeunes diplômés et même professionnels chevronnés afficher la maîtrise voire l'expertise de ces technologies alors qu'en réalité, bon nombre d'entre eux ne maîtrisent que très partiellement ces sujets.

Essayer de déterminer les causes de ce décalage relève davantage de l'essai sociologique que du contexte de cet ouvrage. Mais il y a une leçon à en tirer, une conclusion décisive : les véritables experts sur ces technologies sont rares. Et comme tout ce qui est rare, ils sont précieux. Être le dépositaire de compétences précieuses, c'est bon pour sa carrière. Mieux encore, associer à ces compétences techniques un savoir-faire, une méthodologie de travail cohérente et efficace, c'est disposer d'un facteur différenciant de premier plan, d'un avantage compétitif indiscutable.

C'est précisément l'objectif de cet ouvrage : tenter de vous amener à ce niveau de compétences et de méthode. Des questions suivantes, combien vous laissent indécis, voire vous sont incompréhensibles ?

- XHTML : Connaissez-vous la différence entre `abbr` et `acronym` ? Combien d'éléments `dd` sont possibles pour un même `dt` ? À quoi sert l'attribut `summary` de `table` ? Y a-t-il une différence entre les attributs `lang` et `xml:lang` ? À quoi sert `fieldset` ? Et `tabindex` ?
- CSS : À quelle version appartient `opacity` ? Comment fusionner les bordures des cellules d'un tableau ? Quelle est la différence entre `visibility` et `display` ? Peut-on appliquer une `margin` à un élément `inline` ? Pourquoi définir un `position: relative` sans chercher à repositionner l'élément lui-même ?
- JS : Qu'est-ce que l'*unobtrusive JavaScript* ? Que dire des attributs `href` commençant par `javascript:` ? Qu'est-ce qu'une fermeture lexicale ? Comment déclarer une méthode de classe ? Quels problèmes gravitent autour de la notion de `binding` ?
- DOM : À quoi sert la méthode `evaluate` ? Que se passe-t-il quand on insère un nœud déjà présent ailleurs dans le DOM ? Quel niveau a introduit les variantes NS ? Quelle différence y a-t-il entre nœud et élément ? Une `NodeList` est-elle utilisable comme un simple tableau ?

Ce livre n'apporte pas toutes les réponses à ces questions : s'il devait couvrir exhaustivement toutes ces technologies, vous tiendriez entre les mains un pavé de plus de mille pages, sans doute cher et peu maniable. Mais vous trouverez tout de même de quoi acquérir des bases solides et, surtout, un sens de la qualité, qui vous permettront d'aller plus loin en toute confiance. D'ailleurs, la plupart des chapitres se concluent par une série de ressources, papier ou en ligne, précisément dans cette optique.

Ne faites pas l'impasse sur les chapitres qui vous semblent déjà connus, déjà acquis. Si vous prenez le temps de les lire, j'aime à croire que vous y trouverez au moins quelques concepts ou données techniques qui vous étaient inconnus. Et puis, les sections finales de chaque chapitre sont là pour vous emmener encore plus loin.

Faire la part des choses : Ajax, c'est quoi au juste ?

Le terme « Ajax » est apparu pour la première fois dans un article de Jesse James Garret, sur le site de sa société Adaptive Path, le 18 février 2005 (<http://www.adaptivepath.com/publications/essays/archives/000385.php>). Il s'agit donc d'un terme relativement récent, qui est en réalité l'acronyme de *Asynchronous JavaScript + XML*.

L'article s'intitule *Ajax : une nouvelle approche des applications web*. Il insiste sur le fait qu'Ajax n'est pas une technologie en tant que telle, mais la conjonction de technologies existantes pour une utilisation combinée novatrice. Ici comme ailleurs, le tout est pourtant bien plus grand que la somme des parties.

L'idée de base est la suivante : la plupart des applications web, c'est-à-dire des applications dont l'interface graphique est affichée dans un navigateur, offrent une interaction pauvre et un confort d'utilisation plutôt restreint, en particulier si on les compare aux applications classiques, installées sur les postes utilisateurs. Dans une application web, on reste le plus souvent prisonnier du carcan requête/réponse : pour interagir avec l'application, qu'il s'agisse de valider la saisie de données ou de réorganiser les éléments d'une liste, il faut avancer à petits pas, avec à chaque étape un aller-retour entre notre navigateur et le serveur, qui engendre un rechargement complet de la page.

Par ailleurs, les possibilités offertes à l'utilisateur pour exprimer une demande ou réaliser une action restent primaires : il est rare de voir un site proposer d'ajouter un produit au panier simplement en glissant-déplaçant le premier sur le second. Ces limitations habituelles auraient très bien pu être levées sans utiliser Ajax, comme ce fut d'ailleurs le cas sur une minorité de sites. Toutefois, en bouleversant notre conception de ce qu'il était possible de proposer comme interactions sur une page web, Ajax a naturellement remis sur le devant de la scène la question des interactions riches, notamment au travers du glisser-déplacer et des effets visuels.

Ajax repose sur les technologies suivantes :

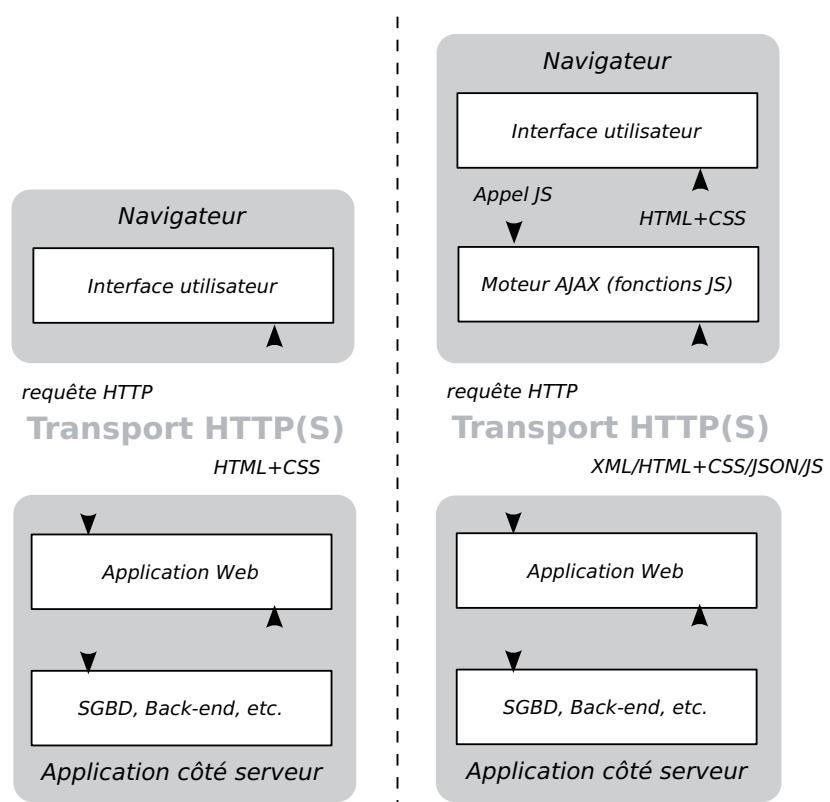
- XHTML pour assurer un balisage sémantique et cohérent du document, ce qui assure que celui-ci sera correctement représenté en mémoire et donc facilement manipulable.
- CSS pour habiller ce balisage sémantique et élargir la gamme des effets visuels utilisables pour communiquer avec l'utilisateur (par exemple, signaler qu'un travail avec le serveur a lieu en arrière-plan, assister un glisser-déplacer, mettre en exergue un élément fraîchement ajouté à une liste).
- DOM pour représenter le document en mémoire, afin d'en manipuler la structure et le contenu sans avoir, justement, à recharger toute la page.

- XML et, dans une moindre mesure, XSLT, pour structurer les données échangées en coulisses entre la page déjà chargée et le serveur, et transformer si nécessaire ces données en contenu affichable.
- XMLHttpRequest, service fourni par Microsoft dans MSIE depuis sa version 5, jusqu'alors méconnu du grand public malgré un intérêt technique qui avait conduit à son implémentation ultérieure par les autres navigateurs. Le premier A de Ajax, c'est lui : le moyen de communication asynchrone.
- JavaScript enfin, qui relie tous ces éléments entre eux.

Là où une page web classique nécessite un rechargement à chaque action, aussi granulaire soit-elle (par exemple, la remontée d'un cran d'un élément dans une liste, afin de l'amener tout en haut de celle-ci), une page exploitant Ajax interagit avec le serveur en coulisses, sans se recharger entièrement. Un code JavaScript réagit à l'action de l'utilisateur en gérant un aller-retour interne avec le serveur, et met éventuellement à jour la page en manipulant directement le DOM de celle-ci. Le schéma suivant compare ces deux approches :

Figure 1-15

Approche traditionnelle et approche Ajax d'un écran d'application web



Vous aurez peut-être remarqué que dans la seconde approche, les données circulant du serveur vers le moteur Ajax sont marquées comme pouvant être non seulement du XML, mais aussi du contenu directement affichable (HTML+CSS), du JavaScript ou des données JSON (*JavaScript Object Notation*, <http://www.json.org>). En effet, rien ne contraint le type des données renvoyées : il appartient au développeur de déterminer ce qui lui semble le plus pragmatique et le plus pratique, au cas par cas. Dans les exemples du chapitre 6, nous mettrons en œuvre plusieurs formats de retour pour illustrer quelques possibilités.

Ce qu'il faut bien comprendre, c'est la nature profondément asynchrone de la communication entre le moteur Ajax et le serveur : pendant que celle-ci a lieu, l'interface utilisateur est toujours présente, et surtout, toujours active. L'utilisateur peut continuer à utiliser la page, en effectuant d'autres actions tandis que la première est en cours de traitement.

Bien entendu, il peut être nécessaire de limiter les actions possibles pendant certains traitements. Par exemple, lorsqu'un produit est en cours de retrait d'une commande, l'utilisateur ne devrait pas avoir la possibilité d'en modifier la quantité. Permettre à l'utilisateur d'effectuer des actions sans attendre la complétion des précédentes impose la mise en place d'indications visuelles de traitement et de garde-fous. Nous verrons des exemples de mise en œuvre au fil des chapitres.

Plan d'action pour deux objectifs : méthode et expertise

Pour tirer le maximum de profit de ce livre, voici un plan d'actions :

- 1 Commencez par l'annexe D pour configurer votre navigateur au mieux, afin de bien suivre les exemples de ce livre et d'augmenter radicalement votre productivité en développement web. Si vous êtes déjà bien à l'aise avec JavaScript, le chapitre 5 peut être utile d'entrée de jeu pour vous permettre de déboguer vos essais plus facilement au fil du livre...
- 2 Si vous n'êtes pas très au point sur les fondamentaux « statiques » (XHTML, CSS), commencez par... les annexes ! Les annexes A et B auront tôt fait de vous (re)mettre en selle. Quant à aller chercher les informations de détail par la suite, rien de tel que l'annexe C pour vous apprendre à naviguer confortablement dans les spécifications. Par la suite, en lisant les chapitres, ne laissez pas une zone d'ombre vous irriter à l'arrière de votre lecture consciente : revenez aux annexes pour trouver l'information ou y piocher les références de ressources détaillées aptes à vous la fournir. La différence entre le bon et l'excellent réside dans la maîtrise des détails autant que dans la vision globale.

- 3 À l'aise sur ces incontournables, prenez le temps de découvrir, ou redécouvrir, JavaScript et le DOM au travers de la première partie et des chapitres 2 à 4. Au-delà de la technique, vous y trouverez de très nombreux conseils et astuces méthodologiques, et la trame d'une démarche de qualité pour votre code futur. Le chapitre 5, lui, vous donnera toutes les clefs d'un débogage de script efficace. N'hésitez pas à faire des exercices, à fouiller les documentations référencées, à faire de nombreux mini-projets ou simples pages de test. Seule la pratique mène à la perfection. Sans une véritable maîtrise de ces composantes, pas de bon développement Ajax possible !
- 4 Vous pouvez ensuite tout naturellement continuer sur Ajax à proprement parler, tant dans son aspect purement « communications asynchrones en arrière-plan » (chapitre 6) qu'au travers de frameworks établis permettant de donner un coup de fouet à votre productivité (chapitre 7) et vous ouvrant les portes d'effets visuels et de comportements impressionnants (chapitre 8). Bien employés, ces derniers donnent des interfaces haut de gamme. Ne laissez toutefois pas l'euphorie technique vous faire oublier l'importance des considérations d'ergonomie et d'accessibilité, qui font toute la différence entre le « peut mieux faire » et le « rien à redire ».
- 5 Pour ouvrir vos horizons et explorer des cas concrets d'utilisation, rien de tel que d'étudier des exemples d'interaction Ajax entre vos pages, éventuellement votre couche serveur, et les services disponibles sur le Web. Apprenez à combiner Ajax, services web, API REST, flux RSS/Atom et *mashups* pour obtenir des interfaces riches en contenu et en fonctionnalités.

Alors, prêts ? Partez !

PREMIÈRE PARTIE

Donner vie aux pages

Ça y est, on se lance. Vous choisissez de ne pas sauter cette partie, et vous avez bien raison. Même si vous pensez en connaître l'essentiel, voire toutes les ficelles, JavaScript et le DOM sont des technologies riches et complexes, dont on n'a généralement pas fait tout le tour, et qu'on a trop souvent découvertes de façon très empirique.

Pourtant, sans une véritable maîtrise de ces technologies, il est difficile de produire des sites Web 2.0 de qualité, ou même simplement robustes et sans bogues.

Le chapitre 2 s'efforce de débarrasser JavaScript de l'image de langage « jouet » dont on l'affuble trop souvent, pour montrer sa richesse et son dynamisme, et mettre en lumière des idiomés avancés et puissants qu'on trouve souvent dans les bonnes bibliothèques de code. Attention toutefois : je ne vais pas détailler les bases de JavaScript (fonctions, variables, boucles, tableaux...) ; si vous ne connaissez pas ces concepts, lisez d'abord un ou deux tutoriels sur les fondamentaux !

Ainsi équipé, vous pourrez comprendre les codes sources qui illustrent le chapitre 3, lequel présente les bases du DOM (d'aucuns trouveront d'ailleurs qu'ils vont bien plus loin que leur idée des bases, ce qui donne une mesure de leur ampleur réelle !). Armé de ces connaissances fiables, vous n'aurez aucun mal à aller fouiller les détails supplémentaires dans les spécifications concernées (et si la forme de ces dernières vous rebute, un petit tour par l'annexe C vous remettra vite en selle).

Afin de vous éviter de devoir gérer les innombrables détails du DOM à la main, ou de réinventer la roue à chaque fonction, le chapitre 4 documente en détail l'extraordinaire bibliothèque Prototype, la plus populaire de l'univers JavaScript. Grâce à elle, vous allez gagner fortement en productivité et en plaisir de développement.

Et pour que ce plaisir ne soit pas gâché par de longues heures passées à s'arracher les cheveux, le chapitre 5 vous donne toutes les billes pour bien déboguer vos scripts.

2

Ne prenez pas JavaScript pour ce qu'il n'est pas

JavaScript est sans doute un des langages les plus incompris de la planète. Tout le monde pense le connaître, croit le maîtriser et le prend néanmoins pour un langage de seconde zone, aux possibilités limitées, même pas capable de faire de l'objet ou de gérer les exceptions !

Rien n'est plus faux. JavaScript est un langage dynamiquement typé doté de la plupart des possibilités usuelles, en dépit d'une syntaxe pas toujours très expressive. De nombreuses possibilités avancées, qui ouvrent les vannes à un véritable torrent de fonctionnalités puissantes, sont souvent mal connues voire inconnues des développeurs web. Et pourtant, JavaScript, que Steve Yegge n'hésite pas à qualifier de *Next Big Language* (le prochain « langage roi »), offre de quoi mettre sur pied des bibliothèques comme Prototype (étudiée au chapitre 4), qui rendent son utilisation quotidienne presque aussi agréable que du scripting Ruby !

Ce chapitre est là pour rendre justice à JavaScript en détaillant certains points souvent mal connus et en faisant la lumière sur certaines fonctionnalités avancées qui restent trop souvent dans l'ombre. À l'issue de ce chapitre, vous serez plus à l'aise pour aller examiner le code de bibliothèques JavaScript avancées, comme Prototype ou script.aculo.us, et créer vos propres bibliothèques haut de gamme.

Mythes et rumeurs sur JavaScript

Commençons par battre en brèche certains mythes autour de JavaScript qui, comme tous les mythes, ont la vie dure.

JavaScript serait une version allégée de Java

Voilà une idée reçue très répandue, qui vient bien entendu de la similarité des noms entre les deux langages. Le créateur du langage l'avait d'abord baptisé LiveScript. Or, en 1995, Netscape sortait la version 2.0 de son navigateur : Netscape Communicator 2.0, qui continuait à pousser en avant Java, fraîchement mis au point par Sun Microsystems, au travers des applets. Netscape 2.0 fournissait également LiveScript, un langage de script censé rendre les pages plus vivantes, et qui permettait notamment de manipuler en partie les applets.

Dans un souci de marketing, le langage de script, dont on ne percevait pas encore l'extraordinaire potentiel, a été renommé JavaScript, et décrit comme un langage « complément de Java » dans un communiqué de presse commun de Netscape et Sun Microsystems : <http://sunsite.nus.sg/hotjava/pr951204-03.html>.

Néanmoins, les deux langages sont très différents. Java est un langage compilé (en code intermédiaire, certes, mais compilé tout de même), avec un système de typage statique, une syntaxe rigoureuse assez verbeuse, et axé sur la représentation de classes et d'objets.

JavaScript, en revanche, est avant tout un langage de script. Cela signifie qu'il est conçu pour être utilisé avec très peu de contraintes et une grande agilité : syntaxe minimaliste et plus flexible, typage dynamique, exécution interprétée, etc. Par ailleurs, en dépit de la présence de concepts objet (objets, instances, champs, méthodes, exceptions, etc.), les aspects plus avancés (héritage, polymorphisme, encapsulation, méthodes abstraites, interfaces, etc.) sont soit absents, soit pris en charge par une syntaxe confuse.

Ce n'est pas que Java est *mieux* que JavaScript, ou inversement : les deux langages répondent à des besoins radicalement différents, sont conçus par des équipes parfaitement distinctes et suivent des évolutions tout à fait autonomes.

JavaScript ne serait basé sur aucun standard

Si JavaScript a vu le jour en tant que projet interne chez Netscape, il a été standardisé très tôt, dès sa version 1.1, par un comité de l'ECMA, organisme de standardisation international, qui continue de le faire évoluer. Le standard ECMA-262 spécifie

EcmaScript, qui représente en quelque sorte le « JavaScript standard ». La seconde édition constitue également un standard ISO (ISO/IEC 16262 pour les curieux).

La troisième édition, qui est aussi la version actuellement finalisée, correspond à JavaScript 1.5 et date de décembre 1999. Tout le travail d'évolution de JavaScript a lieu dans le groupe de travail TG1 à l'ECMA, dirigé par l'inventeur original du langage, Brendan Eich, employé par la fondation Mozilla.

Plusieurs versions intermédiaires ont vu le jour depuis et sont prises en charge notamment par les versions successives de Firefox : JavaScript 1.6 dans Firefox 1.5 (et dans une certaine mesure, Safari 3), JS 1.7 dans Firefox 2, et JS 1.8 dans Firefox 3 ; toutes versions qui incluent des améliorations radicales inspirées d'autres langages de script (générateurs, itérateurs, définition de tableaux par compréhension, affectations multiples, etc.).

JavaScript serait lent

À l'origine, JavaScript était lent, comme l'étaient Java, Perl, Python ou Ruby. Mais la technologie des langages de script et des machines virtuelles a énormément évolué depuis, et tous ces langages sont aujourd'hui relativement rapides. Dans la pratique, on n'a plus de difficulté à faire tourner rapidement des pages applicatives complexes mettant en œuvre de nombreux gestionnaires d'événements, des requêtes Ajax simultanées, et des mises à jour visuelles dues aux règles CSS.

On peut même sortir de la page web pour se pencher sur le navigateur lui-même : l'interface graphique de Firefox, par exemple, est décrite en XUL, un langage basé sur XML, et son fonctionnement est écrit en JavaScript. Même chose pour Thunderbird. Ce n'est d'ailleurs pas sans raison que les fichiers de préférences de ces programmes, nommés `prefs.js`, décrivent les préférences utilisateurs sous forme de code JavaScript. Et pourtant, ces deux programmes sont très confortables à l'utilisation !

JavaScript serait un langage jouet, peu puissant

Ses origines modestes ont enraciné l'image d'un langage jouet, image dont souffrent d'ailleurs presque toujours les langages de script, en particulier aux yeux des aficionados de langages plus rigoureux, plus « sérieux », comme C++ ou Java.

Et pourtant, il y a 10 ans, Java lui-même était dans le camp des « petits », des langages jouets. Davantage d'outils apparaissent chaque jour dans l'univers Linux, qui sont écrits en Python ou en Ruby. Et aujourd'hui, de plus en plus d'applications sont réalisées sur la base de XULRunner, la plate-forme d'exécution de Mozilla basée sur XUL, C++ et JavaScript.

Indépendamment de son utilisation commune, JavaScript n'en est pas moins un langage doté des quelques fonctionnalités critiques aptes à lui permettre de faire de grandes choses. Lesquelles, d'ailleurs, manquent souvent aux langages de meilleure stature. Dans ce chapitre, nous aurons l'occasion d'en découvrir un certain nombre ; attendez-vous à d'agréables surprises...

S'y retrouver entre JavaScript, EcmaScript, JScript et ActiveScript

La compréhension de JavaScript n'est pas facilitée par la pluralité du paysage. On entend parler de JavaScript, de JScript, d'ActiveScript, d'EcmaScript... Comment s'y retrouver ? Voici quelques points de repère.

- JavaScript est le langage d'origine, qui s'appelait LiveScript avant de faire sa sortie publique. Après sa version 1.1, il évolue au sein de l'ECMA, organisme de standardisation au niveau international, sous le nom coquet de ECMA-262, dont la 3^e édition (version actuelle) correspond à JavaScript 1.5. La prochaine édition, originellement prévue pour le deuxième trimestre 2007, constituera JavaScript 2. Ce langage est pris en charge, dans le respect du standard, par la vaste majorité des navigateurs, dont (naturellement) Mozilla, Firefox et Camino, mais aussi Opera, Safari et Konqueror. MSIE prend en charge l'essentiel (voir ci-dessous).
- JScript est le nom donné par Microsoft à son implémentation de JavaScript. Il s'agit grossièrement d'une prise en charge à 95 % de JavaScript, augmentée d'un certain nombre d'extensions propriétaires, par exemple la classe `ActiveXObject` et la méthode `GetObject`. Cette variante n'est, bien entendu, disponible que sur MSIE, et c'est elle qui est documentée dans le MSDN. MSIE 6 fournit JScript 5.6, qui correspond à peu près à JavaScript 1.5.
- ActiveScript n'est pas un langage, mais le moteur d'exécution de scripts, sous Windows, qui permet à une application d'exécuter du code dans plusieurs langages de script, généralement mis à disposition du moteur sous la forme de modules d'extension.

Ainsi, Windows 2000 et Windows XP, qui disposent d'un service nommé WSH (*Windows Scripting Host*), fournissent ActiveScript (c'est pourquoi en double-cliquant sur un fichier .js sous Windows, il tentera d'exécuter le script comme un programme classique ; WSH est une source majeure de failles de sécurité dans Windows...). ASP et ASP.NET utilisent aussi ActiveScript.

Tout ce que vous ne soupçonnez pas : les recoins du langage

Voici le cœur du chapitre. Cette section sert un double objectif :

- 1 Reprendre des aspects du langage souvent traités par-dessus la jambe, et qui sont donc mal maîtrisés, mal connus, entraînant des erreurs d'utilisation par inadvertance et des débogages difficiles.
- 2 Mettre en lumière des aspects souvent inconnus du langage, parfois avancés il est vrai, mais qui constituent les fondations sur lesquelles repose l'extensibilité du langage. Sans ces aspects, des bibliothèques aussi furieusement utiles que Prototype, par exemple, n'auraient jamais pu voir le jour.

Variables déclarées ou non déclarées ?

Vous savez probablement qu'en JavaScript, il est inutile de déclarer les variables. Cette simple affirmation n'est pourtant pas tout à fait exacte. Une variable non déclarée se comporte différemment d'une variable déclarée. Prenons l'exemple de code suivant :

Listing 2-1 Différence entre variables déclarées et non déclarées

```
var total = 0;
var factor = 5;
var result = 42;

function compute(base, factor) {
    result = base * factor;
    factor *= 2;
    var total = result + factor;
    return total;
} // compute

alert('compute(5, 4) = ' + compute(5, 4));
alert('total = ' + total + ' -- factor = ' + factor +
    ' -- result = ' + result);
```

Selon vous, que va afficher ce script ?

Le premier affichage est sans piège : `result` vaut d'abord $5 \times 4 = 20$, `factor` passe à $4 \times 2 = 8$, et `total` vaut $20 + 8 = 28$. On obtient en effet 28.

À présent, le deuxième affichage utilise nos variables `total`, `factor` et `result`. Il s'agit bien sûr des variables déclarées en haut de script, puisque notre affichage est hors de la fonction `compute`, et n'a donc pas accès aux déclarations qui y figurent.

On devrait donc voir s'afficher les résultats de nos affectations : 0, 5 et 42, respectivement. Et pourtant, stupeur : on obtient 0, 5 et 28 ! Que s'est-il passé ?

C'est bien simple : dans une fonction, utiliser une variable sans la déclarer revient à utiliser une variable globale, créée pour l'occasion si besoin. Je dis bien « variable », car les arguments d'une fonction ne sont pas assujettis à cette règle : vous voyez que la modification de `factor`, dans la fonction `compute`, ne touche pas à la variable `factor` déclarée plus haut. On ne modifie que la valeur locale de l'argument `factor` passé à la fonction.

Puisque dans la fonction, `result` n'a pas été déclarée, et qu'une variable externe `result` existe, c'est cette variable qui sera utilisée. En revanche, `total` étant ici déclarée (mot réservé `var`), on obtient une variable locale, et on ne touche pas à la variable externe `total`.

Je vous conseille donc de toujours déclarer vos variables, quitte à le faire à la volée quand la syntaxe le permet, comme pour un index de boucle :

```
for (var index = 0; index < elements.length; ++index)
```

En effet, cela garantit que vous ne touchez pas aux variables globales existantes, et que vous n'avez pas de « fuites », en créant des variables globales inutiles, qui stockent une information normalement interne à votre fonction. Le script suivant illustre bien cette infraction au principe d'encapsulation :

Listing 2-2 Un exemple de « fuite » de donnée par non-déclaration

```
function secretStuff() {
    // traitement confidentiel, avec dedans :
    privateKey = 0xDEADBEEF;
    // fin du traitement
}
secretStuff();
alert(privateKey);
```

Les cas où vos fonctions veulent effectivement manipuler des variables globales sont rares, pour la simple raison que les variables globales sont, dans un code de qualité, rarissimes. Et je ne vous parle pas des oublis de `var` dans des fonctions récursives, c'est un souvenir trop douloureux...

Ceci dit, quand vous en déclarez néanmoins, vous vous demandez peut-être quel intérêt il y aurait à utiliser « `var` » devant le nom, puisqu'on est de toute façon au niveau global ? Il n'y a pas d'intérêt technique clair, mais cela améliorera la lisibilité.

Prenons par exemple le script suivant :

```
MAX = 42;  
total = 0.0;  
count = 0;
```

Comment savoir de façon certaine quelles déclarations constituent des constantes et quelles autres constituent des variables ? Bien sûr, le respect d'une norme de nommage, qui demande généralement qu'on écrive les constantes en majuscules, nous suggère que MAX est une constante, et total et count des variables.

Mais on n'est pas pour autant à la merci d'un développeur peu rigoureux, qui aura juste appelé ses données ainsi sans trop réfléchir, et n'adhère pas à la norme. Cas encore plus fréquent, peut-être MAX a-t-elle été initialement conçue comme une constante, mais au fil du temps la conception a évolué, et aujourd'hui le script la modifie parfois, sans qu'on ait pris la peine de renommer l'identifiant.

Voilà pourquoi des mots réservés comme const et var sont toujours utiles. La version explicite du script ne laisse pas de place au doute, et nécessitera une mise à jour si d'aventure l'évolution du code voulait rendre MAX modifiable :

```
const MAX = 42;  
var total = 0.0;  
var count = 0;
```

Comment, vous ne saviez pas qu'en JavaScript existait le mot réservé const ? Vous voyez que ce chapitre va vous apprendre des choses... Hélas, je suis bien obligé de mettre un bémol : const est une extension à JavaScript 1.5 apportée par Mozilla. Il figure normalement dans les versions 1.6, 1.7 et la prochaine version 2.0, mais pas dans 1.5 et son standard de base, ECMA-262 3^e édition. Aussi, MSIE et Opera ne le prennent pas en charge.

Par conséquent, vous ne trouverez hélas pas de const dans les exemples à venir et l'archive des codes source pour cet ouvrage. Les constantes seront simplement au niveau global, en majuscules, sans var. C'est le mieux qu'on puisse faire pour être portable sur MSIE et Opera. En revanche, si vous gardez le parc client (Mozilla, Firefox, Camino, Konqueror), n'hésitez pas !

Types de données

Certes, JavaScript ne type pas ses variables, arguments et constantes, ce qui ne manque pas d'offenser les partisans des langages statiquement typés, comme C++, Java ou C#. Ceux qui évoluent quotidiennement dans des langages dynamiquement typés, comme JavaScript, Ruby, Python ou Perl, rétorqueront plutôt « et alors ? ».

Que le type ne soit pas déclaré ne signifie absolument pas qu'il n'y a pas de type. En revanche, une variable peut changer de type au gré de ses affectations. Et lorsqu'on tentera d'utiliser une variable en violation de son type actuel, JavaScript nous rappellera à l'ordre en produisant une erreur.

JavaScript dispose tout de même d'un certain nombre de types fondamentaux. Ces types sont plus riches que les descriptions renvoyées pour leurs valeurs par l'opérateur `typeof`, soit dit en passant. Voici un petit tableau récapitulatif pour JavaScript 1.5, disponible à peu près partout.

Tableau 2–1 Types de données de JavaScript et valeurs de `typeof`

Type	<code>typeof</code>	Description
Array	object	Tableau classique, de dimensions quelconques (ex. <code>[]</code> , <code>[1, 2, 3]</code>).
Boolean	boolean	Valeur booléenne : <code>true</code> ou <code>false</code> .
Date	object	Date et heure : <code>new Date(...)</code> .
Error	object	Erreur survenue à l'exécution, généralement capturée par un <code>catch</code> .
Function	object	Cas particulier : on parle ici d'objets fonctions, obtenus en faisant <code>new Function(...){...}</code> . Si on passe à <code>typeof</code> une fonction directement (ex. <code>typeof Math.sqrt</code> , ou encore <code>typeof document.createElement</code>), on obtient logiquement ' <code>function</code> '.
Math	(voir texte)	Math est un singleton (il n'existe qu'un seul objet Math à tout instant, toujours le même) jouant le rôle d'espace de noms : <code>typeof Math</code> renvoie donc ' <code>object</code> '. Pour les objets prédéfinis (tous les autres dans cette liste), <code>typeof</code> renvoie ' <code>function</code> ', car le nom de l'objet est assimilé à son constructeur.
Number	number	Nombre (toujours flottant en JavaScript), équivalent à <code>double</code> (double précision IEEE 754).
Object	object	Un objet quelconque, y compris ceux issus du DOM.
RegExp	function	Une expression rationnelle (type d'objet). Si vous ne savez pas de quoi il s'agit, ou les maîtrisez mal, je ne saurais trop vous recommander d'apprendre (voir bibliographie de fin de chapitre). Disponibles dans pratiquement tous les langages, les expressions rationnelles sont fabuleusement utiles pour le traitement avancé de textes.
String	string	Une chaîne de caractères (type d'objet).

Tout objet dispose de propriétés (ou champs, ou encore attributs : des données dans l'objet) et de méthodes (ou opérations, ou encore fonctions membres : des fonctions dans l'objet). On peut également simuler la notion de propriétés et de fonctions statiques.

Chaque objet a notamment une propriété `prototype`, extrêmement utile, que nous aborderons plus en détail un peu plus loin dans ce chapitre, à la section « Améliorer les objets existants ».

Fonctions et valeurs disponibles partout

JavaScript propose un certain nombre de fonctions globales, accessibles depuis n'importe où. Après une liste succincte, je reviendrai sur quelques points délicats souvent méconnus autour de certaines fonctions.

Tableau 2–2 Fonctions globales de JavaScript

Fonction	Description
Array, Boolean, Date...	Chaque objet prédéfini a une fonction constructeur associée, qui peut assurer certaines conversions appropriées suivant le cas. C'est d'ailleurs pour cette raison que <code>typeof Array</code> et <code>typeof Date</code> renvoient 'function'.
<code>decodeURI</code>	Symétrique de <code>encodeURI</code> (voir plus bas).
<code>decodeURIComponent</code>	Symétrique de <code>encodeURIComponent</code> (voir plus bas).
<code>encodeURI</code>	Encode un URI (une URL, pour simplifier) conformément aux règles d'encodage URL (hormis les minuscules et majuscules non accentuées, les chiffres et certains signes de ponctuation, tout caractère est transformé en séquence hexadécimale %xx voire %uuuu si on est en Unicode). Laisse toutefois le début de l'URL (avant le ? qui marque le début des paramètres) intacte. Par exemple, 'bonjour marc & olivier !' devient 'bonjour%20marc%20&%20olivier%20!'.
<code>encodeURIComponent</code>	Encode un composant d'URI/URL : ne laisse donc aucun caractère spécial intact.
<code>eval</code>	Fonction très importante : elle permet d'exécuter un code JavaScript stocké dans une chaîne de caractères. Cette capacité du langage à s'auto-exécuter est critique, et permet d'avoir des types de réponse JavaScript ou plus spécifiquement JSON dans un contexte Ajax, par exemple. On peut tout imaginer avec cette possibilité : code automodifiant, code délégué, et j'en passe.
<code>isFinite</code>	Permet de déterminer si la valeur stockée dans un <code>Number</code> est finie (<code>true</code>) ou infinie (<code>false</code>). Plus pratique et plus performant que les tests manuels sur <code>Number.NEGATIVE_INFINITY</code> , <code>Number.POSITIVE_INFINITY</code> et <code>Number.NaN</code> , par exemple.
<code>isNaN</code>	Seule manière fiable de détecter qu'un <code>Number</code> ne contient pas une valeur numérique valide (par exemple, s'il a reçu le résultat d'une division par zéro). En effet, la constante <code>NaN</code> (<i>Not a Number</i>) n'est, par définition, égale à aucun <code>Number</code> , pas même à <code>NaN</code> lui-même. Le test (<code>x == NaN</code>) échouera toujours...
<code>parseFloat</code>	Convertit un texte en nombre flottant. Néanmoins, le fonctionnement de la conversion est souvent mal compris, comme nous le verrons plus bas.
<code>parseInt</code>	Convertit un texte en nombre entier. Ne pas préciser la base peut causer des soucis en traitant des représentations utilisant un remplissage à gauche par des zéros, sans parler du mécanisme de conversion qui, comme pour <code>parseFloat</code> , est souvent mal compris (voir plus bas).

On trouve également trois valeurs globales, constantes ou non, dites propriétés globales :

Tableau 2–3 Propriétés globales de JavaScript

Propriété	Description
<code>Infinity</code>	Variable initialement à <code>Number.POSITIVE_INFINITY</code> . Je ne vois pas pourquoi la modifier, aussi préférez les propriétés plus explicites de <code>Number</code> (infinité négative et positive).
<code>NaN</code>	Variable initialement à <code>Number.NaN</code> , qu'on ne modifie normalement jamais ; c'est juste un raccourci.
<code>undefined</code>	Variable équivalente à la valeur primitive du même nom. Permet de simplifier des tests du genre (<code>'undefined' == typeof x</code>) en (<code>undefined === x</code>) grâce à l'opérateur d'égalité stricte.

Les mystères de `parseFloat` et `parseInt`

Toujours indiquer la base, sinon...

Commençons par expliciter le deuxième argument, optionnel, de `parseInt` : l'argument nommé `radix`. Il s'agit de la base numérique pour la conversion. Les valeurs possibles sont :

- 0 (valeur par défaut) pour une « détection » de la base (voir plus bas).
- 2 à 36, qui utilisent les chiffres de 0 à 9 puis autant de lettres de l'alphabet que nécessaire, sans prêter attention à la casse. Ainsi, en base 16 (hexadécimale), on utilise 0-9 et A-F. En base 36, on va jusqu'à Z.

De nombreux développeurs web ne savent même pas que ce deuxième paramètre existe ou ne pensent pas à l'utiliser. Le problème est que, s'il n'est pas précisé, il prend la valeur zéro, et aboutit à une détection automatique de la base selon les premiers caractères du texte :

- Si le texte commence par 0x ou 0X, la suite est de l'hexadécimal.
- S'il commence juste par 0 (zéro), la suite est de l'octal (base 8).
- Sinon, c'est du décimal.

C'est le deuxième cas qui pose régulièrement problème.

Imaginez par exemple que vous ayez un formulaire avec une saisie manuelle de date au format, disons, jj/mm/aaaa. D'ailleurs, un tel texte peut vous être transmis autrement que par une saisie de formulaire... Toujours est-il que vous souhaitez en extraire le mois. La plupart du temps, on procède (à tort) ainsi :

```
var month = parseInt(text.substring(3, 5));
```

Suivant le cas, `text.substring(3, 5)` renverra '01', '02'... '10', '11' ou '12' (pour un numéro de mois valide, en tout cas !). Là-dessus, vous appelez `parseInt` avec simplement cette portion de texte. Pour un mois jusqu'à juillet inclus, ça marchera. Mais août et septembre passent mystérieusement à la trappe ! Vous récupérez alors zéro.

Eh oui ! Faute d'avoir précisé la base (10, dans notre cas), vous avez laissé `parseInt` la détecter, et le zéro initial l'a fait opter pour de l'octal, où seuls les chiffres 0 à 7 sont autorisés. Il convient donc d'être toujours explicite dans l'emploi de `parseInt`, ce qui signifie généralement de mentionner la base 10 :

```
var month = parseInt(text.substring(3, 5), 10);
```

Une conversion plutôt laxiste

Passons à présent à un comportement méconnu commun aux deux fonctions. Tout d'abord, il faut savoir que les espaces préfixes et suffixes sont ignorés, ce qui est en soi pratique :

```
parseInt('25') == parseInt(' 25') == parseInt('25 ') == 25
```

Ce qui peut piéger le développeur est le comportement de ces fonctions en cas de caractère invalide. On a deux cas de figure : un texte invalide dès le premier caractère, et un texte dont le premier caractère est valide.

Dans le premier cas, on récupère automatiquement NaN :

```
var result = parseInt('dommage', 10);
alert(isNaN(result)); // true
```

Dans le second cas, seul le début du texte est utilisé, et le reste est silencieusement ignoré, ni vu, ni connu ! C'est en particulier un problème pour de la validation de saisie, par exemple. Beaucoup de développeurs écrivent un code du type :

```
var value = parseInt(field.value, 10);
// ou parseFloat(field.value), selon ce qu'on veut...
if (isNaN(value))
    // traitement de l'erreur
```

Hélas ! Ça ne suffit pas. Dans un tel contexte, le texte « 42dommage » passera sans problème : on récupérera juste 42. Pourtant, nous souhaitons généralement vérifier que l'ensemble du texte est valide, pas seulement le début !

La solution est relativement simple : il suffit de convertir à nouveau la valeur numérique en texte. Si le résultat est identique au texte d'origine, on est tranquille ! À un détail près toutefois : les espaces de début et de fin, qui ne seront plus là.

Si vous souhaitez effectivement les ignorer, il faudra d'abord les supprimer vous-même, pour avoir un texte de référence comparable à celui que vous obtiendrez par la conversion réciproque. Une telle suppression se fait facilement avec la méthode `replace` des objets `String`. Voici donc une fonction générique de test pour valeurs numériques :

Listing 2-3 Une fonction simple de validation de valeur numérique

```
function isValidNumber(text, intsOnly) {
    text = text.replace(/^\s+|\s+$/.g, '');
    var value = intsOnly ? parseInt(text, 10) : parseFloat(text);
    return String(value) === text;
} // isValidNumber
```

Si l'expression rationnelle employée vous ébranle, je vous encourage encore une fois à apprendre leur syntaxe (par exemple, en consultant <http://www-regexp.com/presentation.php>, qui mis à part sa spécificité PHP, présente les syntaxes universelles). Voici tout de même une explication :

- `^\s+` signifie « un nombre quelconque positif d'espaces en début de texte »
- `|` signifie OU
- `\s+$` signifie « un nombre quelconque positif d'espaces en fin de texte »
- `g` est un drapeau indiquant « remplace toutes les occurrences ». Sans lui, si on retirait des espaces au début, on n'en supprimerait pas à la fin...

On remplace les portions qui correspondent à ce motif par un texte vide, ce qui en pratique élimine les espaces préfixes et suffixes.

Que donne `isValidNumber('42dommage')` ? `intsOnly` n'étant pas spécifié, il vaut `undefined`, ce qui correspond au booléen `false`, et notre opérateur ternaire (`?:`) utilise donc `parseFloat`, lequel renvoie juste `42`. La conversion inverse en texte donne naturellement `'42'`, qui n'est pas égal à `'42dommage'`, donc on retourne `false`.

Et pour un texte vraiment inutilisable ? Par exemple, `isValidNumber('truc', true)` ? Ici on a `intsOnly` à `true`, donc on appelle `parseInt('truc', 10)`, qui renvoie `Nan`. Converti en `String`, cela donne évidemment `'NaN'`, qui n'est pas égal à `'truc'`. On renvoie bien `false`.

Voilà de quoi éviter bien des cauchemars.

Rappels sur les structures de contrôle

Les grands classiques

Vous connaissez normalement les grandes structures de contrôle classiques, qui sont les mêmes en JavaScript qu'en C, C++, Java, PHP et bien d'autres. Voici leurs aspects généraux, qui ne changent strictement rien à vos habitudes :

Tableau 2–4 Aspects généraux des structures de contrôle usuelles

if/else	for	while	do/while	switch/case
if (cond) ...; else if (cond) ...; else ...;	for (init; cond; incr) { ... }	while (cond) { ... }	do { ... } while (cond);	switch (expr) { case expr: ... break; case expr2: ... break; default: ... }

Petite différence entre le `for` JavaScript et le `for` C++/Java ceci dit : si vous déclarez une variable d'index à la volée : `for (var index = ...,` cette variable en JavaScript ne sera pas locale à la boucle : elle existera avec la même portée que la boucle elle-même.

Vous connaissez aussi probablement les traditionnelles instructions `break` et `continue`, également présentes dans les langages cités plus haut : `break` quitte la boucle courante, tandis que `continue` saute immédiatement au tour suivant (ou quitte la boucle si c'était son dernier tour).

Labélisation de boucles

En revanche, tous les langages ne permettent pas à `break` et `continue` d'être *labélisés*. Il existe en effet une syntaxe qui permet d'affecter un nom à une boucle. L'avantage est qu'on peut alors fournir ce nom à un `break` ou un `continue`, qui vont fonctionner au niveau de la boucle ainsi nommée, plutôt qu'au niveau de la boucle courante. On peut donc sortir de, ou court-circuiter, plusieurs niveaux de boucles imbriquées.

Voici un exemple de recherche dans un cube de données, qui permet de quitter les trois niveaux de boucle dès que l'élément cherché a été trouvé (ce qui est largement préférable au fait de laisser les trois boucles se dérouler !), sans avoir à gérer un booléen `found` supplémentaire.

Listing 2-4 Un exemple pertinent de break labélisé

```
function findAndShow(cube, value) {
    var coords = null;
outerLoop:
    for (var x = 0; x < cube.length; ++x)
        for (var y = 0; y < cube[x].length; ++y)
            for (var z = 0; z < cube[x][y].length; ++z)
                if (cube[x][y][z] == value) {
                    coords = [x, y, z];
                    break outerLoop;
                }
    alert(coords
        ? 'Trouvé en ' + coords.join(',')
        : 'Pas trouvé');
} // findAndShow
```

for...in, la boucle si souvent mal employée

Une boucle JavaScript mal connue, ou mal comprise, est le `for...in`. Sa syntaxe est très simple :

```
for (variable in object)
    ....
```

Contrairement à une idée reçue, ou simplement à l'intuition, cette boucle ne permet pas (hélas !) d'itérer sur les éléments d'un tableau ! Non, son rôle est tout autre : elle itère sur les propriétés de l'objet.

On peut donc par exemple obtenir une liste des noms des propriétés d'un objet quelconque avec le code suivant :

Listing 2-5 Énumération des méthodes d'un objet avec for...in

```
function showProperties(obj) {
    var props = [];
    for (var prop in obj)
        props.push(prop);
    alert(props.join(', '));
} // showProperties
```

On utilise ici un aspect peu connu de JavaScript, que nous reverrons plus tard : les opérateurs `[]` et `.` sont presque synonymes. Pour accéder à une propriété d'un objet `obj`, alors que le nom de la propriété est stocké dans une variable `prop`, on ne peut pas faire `obj.prop` (cela chercherait une propriété nommée « `prop` »), mais `obj[prop]` fonctionne !

Ainsi, le code suivant :

```
| showProperties(['christophe', 'élodie', 'muriel']);
```

produira l'affichage suivant :

```
| 0, 1, 2
```

Mais... De quoi s'agit-il ? En fait, sur un tableau, les propriétés détectables sont les indices existants du tableau. On a ici trois éléments, donc les indices 0, 1 et 2. Cet exemple n'est pas très convaincant, et nous n'aurons pas plus de chance sur tous les objets natifs de JavaScript : nous n'aurons que les propriétés (ou méthodes) ajoutées par extension de leur prototype, donc faute de code préalable, on n'obtiendra rien.

Tentons plutôt un saut en avant vers le chapitre 3, consacré au DOM, et testons ceci :

```
| showProperties(document.implementation);
```

Sur un navigateur conforme aux standards, on obtiendra :

```
| hasFeature, createDocumentType, createDocument
```

Ah ! Voilà qui est plus sympathique. Ce sont bien en effet les trois méthodes prévues par l'interface `DOMImplementation`, proposées par l'objet `document.implementation`. Notez qu'on a obtenu des méthodes, pas de simples champs. Les méthodes constituent elles aussi des propriétés.

Employer `with`, mais avec circonspection

Pour terminer, précisons une dernière structure de contrôle fort pratique dans certains cas, mais parfois source de confusion, qu'on retrouve également en Delphi (bien que les deux langages soient sans aucun rapport !) : le `with`.

Ce mot réservé permet d'étendre la portée courante en lui ajoutant celle d'un objet précis. Qu'est-ce que la portée ? Disons qu'il s'agit de l'ensemble des endroits où l'interpréteur va rechercher un identifiant. Par exemple quand vous tapez :

```
| alert(someName);
```

Où JavaScript va-t-il chercher `someName` ? Par défaut, comme beaucoup de langages, il suit le chemin classique : le bloc courant, son bloc parent, et ainsi de suite jusqu'à la fonction. S'il est dans une méthode, il cherchera alors dans l'objet conteneur. Puis dans les éventuels espaces de noms contenant l'objet. Et enfin au niveau global. S'il ne trouve rien, vous obtiendrez une erreur.

Étendre la portée en lui rajoutant le contexte d'un objet précis est très utile pour simplifier ce genre de code :

Listing 2-6 Un exemple typique de code qui bénéficierait d'un with...

```
field.style.border = '1px solid red';
field.style.margin = '1em 0';
field.style.backgroundColor = '#fdd';
field.style.fontSize = 'larger';
field.style.fontWeight = 'bold';
```

Alors qu'avec `with`, ça donne :

Listing 2-7 Le même avec un with judicieusement employé

```
with (field.style) {
    border = '1px solid red';
    margin = '1em 0';
    backgroundColor = '#fdd';
    fontSize = 'larger';
    fontWeight = 'bold';
}
```

Pratique, n'est-ce pas ? En effet, mais il vaut mieux restreindre ces usages à de tout petits segments de code : la tentation est grande de l'employer à tort et à travers pour s'épargner de la frappe, et aboutir à un code où on ne sait plus trop, à la lecture, quelle propriété ou méthode appartient à quel objet ! En tout état de cause, évitez absolument *d'imbriquer* les `with` : la lisibilité en pâtit de trop, et ce serait probablement une source d'erreurs.

Opérateurs méconnus

JavaScript dispose de nombreux opérateurs, dont les grands classiques bien sûr, mais aussi certains moins connus. Il gère également certains opérateurs classiques d'une façon particulière.

Retour rapide sur les grands classiques

JavaScript reprend la plupart des opérateurs classiques, mais on a aussi quelques opérateurs moins courants, même s'ils existent parfois dans d'autres langages.

Tableau 2–5 Opérateurs classiques également présents dans JavaScript

Catégorie	Opérateurs
Arithmétiques	+, -, *, /, %, ++, --, - unaire (inversion de signe)
Affectations	=, +=, -=, *=, /=, >>=, <<=, &=, =, ^=
Bit à bit	&, , ^, ~, <<, >>
Comparaisons	==, !=, >, >=, <, <=
Logiques	&&, , !
Textuels	+ et += pour la concaténation
Accès aux membres	obj.propriété et obj["propriété"] (déjà évoqués plus haut)

Opérateurs plus exotiques

Par exemple, l'opérateur `===` (sa négation étant `!==`) représente l'égalité stricte. En plus d'être égaux en valeur, les opérandes doivent avoir le même type. On le trouve aussi, par exemple, dans PHP. Ainsi, `'3' == 3`, mais `'3' !== 3`.

L'opérateur `in` permet de déterminer si son opérande de gauche est bien une propriété de l'objet opérande de droite. C'est très pratique pour éviter les cas délicats de méthodes plus générales. Ainsi, on a l'habitude de tester que l'objet global `document` a bien une méthode `createTextNode` en faisant simplement ceci :

```
| if (document.createTextNode)
```

C'est correct, car si la méthode manque, l'expression vaut `undefined`, équivalent ainsi à `false`, et le test échoue. Mais *quid* d'un test sur une propriété booléenne, ou simplement pouvant valoir `null` ? Par exemple, considérez le test suivant :

```
| if (someDOMNode.firstChild)
```

Ce test ne permet pas de savoir si `someDOMNode` a bien une propriété `firstChild` : il suffit que `someDOMNode` soit un élément vide pour qu'il ait bien cette propriété, mais qu'elle vaille `null`, faisant échouer le test.

L'opérateur `in` à la rescousse !

```
| if ('firstChild' in someDOMNode)
```

Notez que l'opérande de gauche est une valeur (un nombre, un texte, etc.) qui correspond au nom de la propriété.

Comportements particuliers

En JavaScript, n'importe quelle valeur possède un équivalent booléen, pour pouvoir être utilisée dans le cadre d'une expression de test. La règle est simple : les valeurs `null`, `undefined`, `false`, `-0`, `+0`, `Nan` et `''` (chaîne vide) équivalent à `false`. Le reste équivaut à `true`.

Si les zéros signés vous étonnent, souvenez-vous que les nombres JavaScript sont des nombres flottants conformes au standard IEEE 754 : on gère le zéro positif et le zéro négatif. Quelle différence ? Un zéro positif est obtenu, par exemple, en divisant un nombre par un infini de même signe, tandis qu'un zéro négatif est obtenu en divisant un nombre par un infini de signe opposé.

C'était la minute culturelle, parce que franchement, le jour où vous aurez besoin de cette distinction dans vos scripts...

Il est important de savoir qu'il est donc possible de tout utiliser dans un test et avec les opérateurs logiques. Il faut juste bien se souvenir quelles valeurs équivalent à `false`. Heureusement, la liste est intuitive.

Prise en charge des exceptions

Depuis sa version 1.4, JavaScript gère les exceptions, ce qui en améliore considérablement la robustesse. JScript a attendu sa version 5.6 (MSIE 6).

Rappelons rapidement qu'une exception, c'est une erreur d'exécution. Une situation exceptionnelle (car il ne devrait pas y avoir d'erreur, pardi !) dans laquelle le programme est soudain plongé. Expliquer dans le détail le bien-fondé des exceptions par rapport aux mécanismes plus anciens de signalement et de traitement des erreurs sort largement du cadre de ce chapitre. Si le sujet vous intéresse, consultez le site suivant : http://fr.wikipedia.org/wiki/Système_de_gestion_d'exceptions.

Les types d'exceptions prédefinis

Bien qu'on puisse utiliser absolument toutes les expressions comme représentation de l'erreur, JavaScript définit tout de même, depuis la version 1.5, six types d'erreurs précis. Il s'agit de spécialisations de l'objet `Error`, lequel fournit principalement des propriétés `name` et `message`. La première fournit le nom de l'erreur, c'est-à-dire le nom de son type ; la seconde fournit le message passé au constructeur de l'objet.

Parmi ces types, on peut imaginer réutiliser `TypeError` et surtout `RangeError` dans nos propres scripts. Le reste exprime surtout des situations survenant dans les fonctions globales.

Tableau 2–6 Types prédéfinis d'erreur en JavaScript 1.5

Type	Description
EvalError	Problème à l'exécution du code fourni à eval.
RangeError	Indique qu'un argument ou une variable avait une valeur hors de l'intervalle autorisé.
ReferenceError	Utilisation d'un identifiant inconnu (faute de frappe, etc.).
SyntaxError	Problème syntaxique à l'analyse du code fourni à eval.
TypeError	Indique qu'un argument ou une variable avait un type invalide. Par exemple, méthode inconnue.
URIError	Erreur de syntaxe dans un URI passé à encodeURI ou decodeURI.
InternalError	Erreur de l'interpréteur. Par exemple, récursion infinie détectée ! N'existe pas sur certains navigateurs...

Capturer une exception : try/catch

Lorsque vous souhaitez capturer une exception susceptible de survenir dans un bloc de codes, vous devez encadrer ce bloc par un `try...catch`, selon le schéma suivant :

```
try {
    // code susceptible de lancer l'exception
} catch (e) {
    // traitement de l'exception, stockée dans l'objet e
}
```

Le principe, identique au traitement dans les autres langages gérant les exceptions, est le suivant : si le code encadré par `try` et `catch` ne lève aucune exception, le contenu du `catch` est ignoré, et l'exécution se poursuit sur la ligne suivant l'accolade fermante. En revanche, si une exception survient n'importe où dans le code encadré, le reste de ce code est court-circuité, et le `catch` est déclenché. L'exception est rendue accessible au travers de la variable nommée entre les parenthèses. Vous pouvez ainsi examiner son nom, son message et son type.

Dans la pratique, il est rare de voir survenir des exceptions issues du système, ou du navigateur lui-même. La plupart du temps, vous lancerez les exceptions vous-même, et les rattraperez plus haut dans la pile d'appels.

Voici tout de même un exemple impliquant des exceptions natives, qui suppose que l'utilisateur a pu taper un code JavaScript dans un champ de saisie d'ID `jsCode`, et nous a demandé de l'exécuter (ce qui, en dehors d'un exemple didactique, constituerait un trou de sécurité béant !) :

Listing 2-8 Un exemple de traitement d'exceptions natives

```

function evalCode() {
    var code = document.getElementById('jsCode');
    var errorZone = document.getElementById('error').firstChild;
    errorZone.nodeValue = '';
    try {
        var result = eval(code.value);
        if (undefined !== result)
            alert('Résultat : ' + result);
    } catch (e) {
        if (e instanceof SyntaxError)
            errorZone.nodeValue = 'Erreur de syntaxe : ' + e.message;
        else if ('undefined' != typeof InternalError
            ↪ && e instanceof InternalError)
            errorZone.nodeValue = 'Erreur interne : ' + e.message;
        else
            errorZone.nodeValue = e.name + ' : ' + e.message;
    }
    code.focus();
} // evalCode

```

Observez l'utilisation de l'opérateur `instanceof`, qui permet de déterminer si l'objet en opérande gauche est compatible avec le type en opérande droit (s'il est de ce type ou d'un type descendant, donc). C'est ainsi qu'on distingue entre les différents types d'erreurs, en JavaScript. Afin que le code passe sur tous les navigateurs, on vérifie aussi que le type `InternalError` est bien reconnu.

Vous trouverez l'exemple complet dans l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles. Si vous utilisez Konqueror, vous n'obtiendrez que les erreurs de syntaxe : Konqueror utilise une gestion particulière, à base de « rapports d'erreurs », pour les autres cas.

Voici quelques exemples d'affichages produits par notre script :

Tableau 2-7 Exemples de comportement de notre script

Code saisi	Résultat (Firefox 1.5)
5 + 3	Boîte de message « Résultat : 8 »
x + 3	ReferenceError : x is not defined
5.times(3)	Erreur de syntaxe : missing ; before statement
(5).times(3)	TypeError : 5.times is not a function
function f() { f(); }	Erreur interne : too much recursion
f();	

Voilà donc de quoi gérer aussi bien les erreurs natives que celles que nous pourrions signaler en lançant nous-mêmes une exception (ce que nous apprendrons à faire plus loin).

Extension : catch multiples et conditionnels

Les navigateurs Netscape 6+ et Mozilla (Mozilla, Firefox, Camino) proposent une extension à cette syntaxe pour JavaScript 1.5, que je signale par souci d'exhaustivité, mais qui doit être utilisée uniquement lorsque l'on possède la maîtrise de l'environnement client (pour un intranet avec des postes standardisés, par exemple).

L'extension permet d'obtenir une syntaxe similaire à celle du `try/catch` dans la plupart des langages, en ayant plusieurs blocs `catch` afin de différencier plus clairement les comportements, en général selon le type de l'exception. Voici ce que cela donnerait pour notre exemple précédent :

Listing 2-9 La fonction précédente, reformulée en catch multiples

```
function evalCode() {
    var code = document.getElementById('jsCode');
    var errorZone = document.getElementById('error').firstChild;
    errorZone.nodeValue = '';
    try {
        var result = eval(code.value);
        if (undefined !== result)
            alert('Résultat : ' + result);
    } catch (e if e instanceof SyntaxError) {
        errorZone.nodeValue = 'Erreur de syntaxe : ' + e.message;
    } catch (e if e instanceof InternalError) {
        errorZone.nodeValue = 'Erreur interne : ' + e.message;
    } catch (e) {
        errorZone.nodeValue = e.name + ' : ' + e.message;
    }
    code.focus();
} // evalCode
```

C'est sympathique, mais cela n'ajoute rien d'un point de vue fonctionnel : il s'agit juste de sucre syntaxique. Aussi, je vous conseille de vous en abstenir pour des pages n'ayant pas la certitude d'être vues sous un navigateur Netscape 6+ ou Mozilla.

Point important : une clause `catch` ne peut capturer que les exceptions survenues dans le bloc `try`, pas dans un autre `catch` ! Celles-ci se propagent normalement.

Garantir un traitement : finally

Le `try/catch` est bien pratique, mais il lui manque une alternative. En effet, imaginons que nous souhaitions seulement capturer une partie des exceptions possibles

(voire aucune), mais garantir qu'un traitement donné soit exécuté en fin de code dangereux.

Les exemples ne manquent pas, car ce besoin apparaît à partir du moment où on alloue une ressource de façon temporaire : fichier, *socket*, mémoire, etc. On souhaite l'ouvrir ou l'allouer, la manipuler et assurer sa fermeture ou libération, qu'on ait rencontré une erreur ou non.

Ce besoin fréquent est la raison d'être de la clause `finally`. Cette clause peut être ajoutée au `try`, après tous les éventuels `catch`. Il ne peut y en avoir qu'une. Un `try` a donc au moins un `catch` ou un `finally`. Le contenu de cette clause est exécuté quoi qu'il arrive.

On peut imaginer un code du genre :

```
var res = new HeavyResource(args);
try {
    // Code manipulant res, qui a potentiellement des erreurs
} finally {
    delete res;
}
```

Qu'il y ait des erreurs ou pas, on libère la ressource. C'est très pratique (et cela manque un peu au C++, par exemple, qui doit recourir à des astuces de *smart pointers* pour obtenir le même résultat).

Lancer sa propre exception : `throw`

Tout l'intérêt du mécanisme des exceptions est de nous permettre de signaler une erreur à un endroit du code qui n'est pas à même de la résoudre ; plus haut dans la pile d'appels, un autre contexte contiendra peut-être les informations nécessaires pour apporter un traitement satisfaisant, ce qu'on ne manquera pas de faire à l'aide d'un `try/catch` prévu pour capturer l'erreur en question.

Seulement voilà, comment la lance-t-on, notre erreur ? À l'aide de l'instruction `throw`. Celle-ci prend un argument unique, qui est l'objet représentant votre erreur. Il peut s'agir de n'importe quel objet, ce qui en JavaScript signifie n'importe quelle valeur : 42, 'Dommage Éliane !', un objet à vous ou une instance d'erreur prédefinie feront tout autant l'affaire.

Évidemment, si vous souhaitez pouvoir apporter un traitement générique des erreurs (pour maintenir un journal d'erreurs par exemple), il vaut mieux fournir à chaque fois des objets compatibles avec `Error` : des instances d'erreurs prédefinies (voir tableau 2-6), ou d'objets qui vous appartiennent et qui sont extérieurement compatibles avec `Error` (c'est-à-dire ayant des propriétés `name` et `message`). Voici un exemple :

Listing 2-10 Exemples de lancements d'exceptions

```
function CustomError(message, info) {
    this.name = 'CustomError';
    this.message = message;
    this.customInfo = info;
} // CustomError

try {
    // Code qui s'exécute
    throw new CustomError('souci !', 42);
    // Code court-circuité
} catch (e) {
    if (e instanceof CustomError)
        alert(e.message + ' / ' + e.customInfo);
    else
        throw e;
}
```

Notez que puisqu'on utilise ici une fonction constructeur, l'opérateur `instanceof` fonctionne correctement.

Vous voyez aussi une utilisation courante de `throw` : la repropagation d'une exception depuis un `catch`. En effet, un `catch` « tue » l'exception, qui ne se propage plus : elle est morte, pour ainsi dire. Il est toutefois possible de la relancer, puisque `throw` lance son argument comme exception. Une clause `catch` peut donc simplement « intercepter » une erreur, par exemple pour la consigner dans un journal d'erreurs.

Améliorer les objets existants

JavaScript est un langage à prototype (par opposition à un langage basé sur les classes et sous-classes pour réaliser l'héritage). Tout objet JavaScript est doté d'une propriété `prototype`, qui représente le modèle sur lequel l'objet en question se base. Un objet dispose automatiquement de toutes les propriétés de son prototype. JavaScript permet de modifier le prototype d'un objet à n'importe quel moment, et d'ajouter ou d'enlever des propriétés à tout instant.

Un peu de théorie sur les langages à prototypes

C'est une notion très déroutante pour les personnes habituées à un système de classes et d'instances, comme Delphi, C++, Java ou C#. Il est inutile d'en maîtriser tous les détails pour écrire la très vaste majorité des scripts, mais cela aide à comprendre le code de bibliothèques riches, comme celui de Prototype (le bien nommé), par exemple.

Voici un petit résumé des principales différences, pour les curieux :

Tableau 2–8 Différences entre langages à classes et à prototypes

Dans un langage à classes...	Dans un langage à prototypes...
On distingue les classes et leurs instances : les objets.	Tout est une instance.
On définit une classe avec une définition, on l'instancie avec une méthode constructeur.	On définit et on instancie des objets à l'aide d'une fonction constructeur (voir <code>CustomError</code> dans le listing 2-10).
On obtient une hiérarchie en référençant la classe mère dans la définition de la classe fille.	On obtient une hiérarchie en définissant un objet père comme prototype de la fonction constructeur fille.
L'héritage suit la chaîne de classes.	L'héritage suit la chaîne de prototypes.
Les définitions d'une classe et de ses classes ancêtres fournissent l'ensemble complet et définitif des propriétés de toutes les instances de la classe.	Les fonctions constructeurs définissent un jeu initial de propriétés. On peut ajouter ou ôter des propriétés à tout moment pour un objet individuel, ou pour tous les objets basés sur un même prototype.

JavaScript est donc particulièrement flexible quant aux définitions des objets, ce qui permet d'obtenir certains idiomés très pratiques, comme les objets anonymes, ou la simulation d'héritage statique par recopie des propriétés d'un prototype dans un autre (au cœur de la bibliothèque Prototype actuelle).

Mais si tout ceci vous plonge dans la plus extrême confusion, soyez sans crainte : vous n'avez pas besoin de trouver le tableau 2-8 d'une limpide cristalline pour comprendre ce qui va suivre.

Retenez simplement qu'en modifiant les propriétés (champs ou méthodes) du prototype d'une classe (techniquement, il faudrait écrire « *d'une fonction constructeur* », mais quand je dis « *classe* », avouez que cela vous paraît plus clair, non ?), on affecte toutes les instances de cette classe (tous les objets créés par cette fonction constructeur). Et il faut noter que l'on affecte aussi ceux créés avant la modification ! En effet, ils référencent tous le même prototype.

Si vous voulez réellement réfléchir en termes classiques (c'est le cas de le dire), vous pouvez voir cela comme des instances qui feraient toutes de la composition/délégation dynamique sur un singleton.

Mise en pratique

Ce mécanisme nous permet d'augmenter les possibilités d'objets prédéfinis. Vous venez peut-être de Java, et êtes tout dépité de ne trouver dans `String` ni `startsWith` ni `endsWith`, sans parler de `trim`? Il vous manque des méthodes dans `Array` (ah bon ? Vous êtes sûr d'avoir bien regardé, parce qu'il y en a un paquet...) Qu'à cela ne tienne, rajoutez-les !

Nous allons voir ensemble deux exemples. Nous allons d'abord étendre `String`, en lui ajoutant les trois méthodes que nous venons d'évoquer (et même une quatrième, tant qu'à faire !) Ensuite, nous ajouterons à `Array` une des rares fonctionnalités qui lui manquent vraiment.

Étendre ses Strings

Voici comment étendre les possibilités de n'importe quelle valeur représentée par JavaScript à l'aide d'un objet `String` (y compris tout ce qui vient du DOM !).

Listing 2-11 Extensions sympathiques de String par son prototype

```
String.prototype.endsWith = function(suffix) {
    return this.length - suffix.length == this.lastIndexOf(suffix);
};

String.prototype.isBlank = function() {
    return null != this.match(/^\s*$/);
};

String.prototype.startsWith = function(prefix) {
    return 0 == this.indexOf(prefix);
};

String.prototype.trim = function() {
    return this.replace(/^\s+|\s+$/g, '');
};
```

Toutes les chaînes de caractères, peu importe leur provenance et leur date de création, disposent maintenant de quatre fonctions supplémentaires. Nous verrons un peu plus loin un exemple d'exécution.

Array et les injections

L'objet `Array` de JavaScript est sans doute l'un des plus méconnus. La plupart des gens se bornent à utiliser ses propriétés d'index (`data[0]`, etc.) et sa propriété `length`, pour boucler sur les éléments par exemple. Pourtant, en tant qu'objet, il dispose de nombreuses méthodes très pratiques, qui nous facilitent considérablement l'écriture du code.

Rien que pour vous donner une idée, voici leur liste à partir de JavaScript 1.6, sans entrer dans les détails : `concat`, `every`, `filter`, `forEach`, `indexOf` (si !), `join`, `lastIndexOf`, `map`, `pop`, `push`, `reverse`, `shift`, `slice`, `some`, `sort`, `splice`, `toSource`, `toString`, `unshift` et `valueOf`. Je suis prêt à parier qu'au moins certaines vous sont inconnues, en particulier les méthodes d'itération : `every`, `filter`, `forEach`, `map` et `some`. Il faut avouer qu'elles sont apparues avec JavaScript 1.6, et MSIE ne supportant déjà pas tout le standard 1.5...

Toutefois, même sur un navigateur prenant en charge toutes ces méthodes (les navigateurs Mozilla typiquement), il manque une fonction courante des objets énumérables : l'injection.

Pour simplifier, un énumérable est une séquence d'éléments, sur laquelle il est possible d'itérer du premier élément jusqu'au dernier. Une injection sur un énumérable consiste à définir une valeur, dite accumulateur, avec une valeur initiale, et qui va évoluer en lui appliquant une fonction donnée pour chaque valeur de l'énumérable.

Par exemple, supposons un accumulateur de valeur initiale zéro, et une fonction d'injection qui, en recevant l'accumulateur et l'élément courant de l'itération, renvoie la somme des deux. En injectant cette fonction sur un énumérable de nombres, on obtient la somme. Un accumulateur démarrant à 1 (un) et une fonction renvoyant le produit, retournent le produit interne de l'énumération, etc.

C'est un mécanisme très utile, surtout si l'on ajoute une astuce : faute de valeur explicite pour la valeur initiale de l'accumulateur, on prend le premier élément de l'énumérable, et on itère à partir du second.

Voici le code JavaScript qui ajoute cette possibilité aux tableaux, en montrant tant qu'à faire deux utilisations communes de l'injection :

Listing 2-12 Ajout de l'injection aux tableaux, deux utilisations courantes

```
Array.prototype.inject = function(fx, acc) {
    if (0 == this.length)
        return acc;
    var start = undefined === acc ? 1 : 0;
    if (undefined === acc)
        acc = this[0];
    for (var index = start; index < this.length; ++index)
        acc = fx(acc, this[index]);
    return acc;
} // inject

Array.prototype.sum = function() {
    return this.inject(function(acc, num) {
        return acc + num;
    });
} // sum

Array.prototype.average = function() {
    return this.sum() / this.length;
} // average
```

Remarquez que le cas du tableau vide ne conduit à aucune erreur dans `average : sum` renverra `undefined`, `length` renverra zéro, et `undefined / 0` donne `Nan`, ce qui est plutôt bien pour la moyenne d'un tableau vide, qui n'a donc aucun élément pour produire une moyenne valide.

Nous venons de voir l'aspect théorique, mais que cela donne-t-il en pratique ? Dans l'archive des codes source pour ce livre, disponible sur le site des éditions Eyrolles, vous trouverez un exemple complet, dont le résultat est présenté à la figure 2–1.

Figure 2–1
Exécution d'exemples utilisant ces extensions



Plutôt chouette, vous ne trouvez pas ? Les possibilités sont infinies... Enfin, pas tout à fait infinies quand même. Autant dans Mozilla, Firefox et Camino, tout objet a un prototype, autant d'autres navigateurs limitent cet aspect aux objets issus de fonctions constructeur. Ainsi, les objets du « DOM navigateur » (`window`, `navigator`, etc.) et les objets du DOM standard n'ont pas forcément de prototype. Le code suivant ne fonctionnera pas partout, hélas !

```
HTMLElement.prototype.myInnerText = function() {  
    ...  
}
```

Arguments des fonctions

JavaScript fait partie de ces langages qui n'exigent pas une concordance entre le nombre de paramètres et celui des arguments. Qu'une fonction déclare ou non des paramètres (en fournissant leurs noms entre parenthèses), il est possible de lui passer des arguments. Déclarer des paramètres ne sert qu'à donner un nom aux premiers arguments transmis. On peut en passer plus, moins, voire aucun.

Tout paramètre déclaré, pour lequel aucun argument n'est transmis, a la valeur spéciale `undefined`. Il existe deux moyens de tester cet état. Le plus long, et pourtant le plus répandu :

```
| ('undefined' == typeof paramName)
```

Le plus court, qui est d'ailleurs le plus performant aussi, est moins répandu (vraisemblablement par manque d'information) :

```
| (undefined === paramName)
```

Quant aux arguments passés en surplus, il faut y accéder d'une façon plus générique. En effet, chaque fonction a automatiquement une variable locale nommée `arguments`. Cette variable n'est pas à proprement parler un tableau (ce n'est pas toujours une instance de `Array`), mais elle se comporte comme tel, en fournissant une propriété `length` et des propriétés d'`index`, qui permettent donc la notation `arguments[numéro]`, en débutant bien entendu à zéro.

Voici par exemple une fonction qui affiche l'ensemble de ses arguments. Dans la mesure où `arguments` n'est pas forcément un `Array`, on va créer le texte à afficher manuellement puisqu'on n'a pas de garantie d'existence d'une méthode `join`. On verra toutefois à la prochaine section comment utiliser `join` malgré cette incertitude.

Listing 2-13 Une fonction affichant la liste de ses arguments

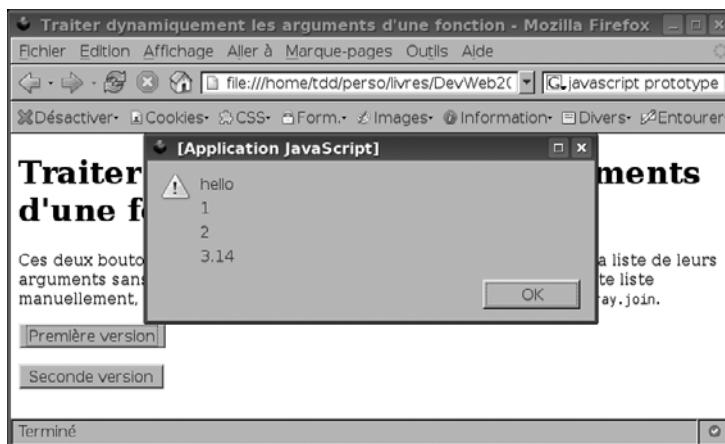
```
function showArgs() {
    var msg = '';
    for (var index = 0; index < arguments.length; ++index)
        msg += arguments[index] + '\n';
    alert(msg);
} // showArgs

showArgs('hello', 1, 2, 3.14);
```

L'affichage obtenu donne ceci (la page d'exemple, fournie dans l'archive des codes source, illustre aussi la section suivante) :

Figure 2-2

Notre fonction affiche bien ses arguments !



Cette variable `arguments` permet de réaliser facilement des fonctions à paramètres variables. Pour les habitués de C ou Java 5 : c'est comme la notation ... de ces langages. Voici par exemple une fonction qui prend un facteur en argument, puis un nombre quelconque de valeurs, et renvoie le tableau des produits.

Listing 2-14 Un exemple de fonction à paramètres variables

```
function multiply(factor) {
    var result = [];
    for (var index = 1; index < arguments.length; ++index)
        result.push(factor * arguments[index]);
    return result;
} // multiply
```

Le binding des fonctions : mais qui est « `this` » ?

Le *binding* de fonction est un sujet très souvent mal compris par les développeurs JavaScript, et cause d'un nombre incalculable de questions sur les forums. Techniquement, il n'est pourtant pas complexe.

Tout d'abord, il faut bien comprendre que le sujet n'a d'intérêt que lorsqu'on utilise une fonction en tant que *méthode*, c'est-à-dire membre d'un objet qui utilise d'autres propriétés/méthodes du même objet.

Pour avoir accès à ces autres membres du même objet, le code doit explicitement qualifier ces membres en indiquant leur objet contexte, à l'aide du mot réservé `this`. En effet, `this` référence toujours « l'objet courant ». C'est justement cette notion d'objet courant, ou contexte, qu'on appelle le *binding*.

Dans la plupart des langages orientés objet, le *binding* est trivial : il est défini lorsqu'on appelle une méthode en précisant l'objet sur lequel on l'appelle, généralement à l'aide de l'opérateur point (.) ou flèche (->) ; les appels de méthode implicites qui ont lieu ensuite conservant ce contexte. Par exemple, en Java on peut écrire :

Listing 2-15 Binding explicite et implicite en Java

```
class MyClass
{
    void doSomething() {
        // ...
        doMoreStuff();
    }
    void doMoreStuff() { ... }
}
MyClass myObj = new MyClass();
myObj.doSomething();
```

L'appel en bas de listing *explicite le contexte* en précisant l'objet sur lequel la méthode est appelée (`myObj`). Ce contexte est valide jusqu'à nouvel ordre, ce qui fait que dans `doSomething`, on peut simplement appeler `doMoreStuff()`, sans avoir à faire `this.doMoreStuff()` (qui resterait valide mais serait inutilement verbeux).

De nombreux langages sont ainsi faits, comme Delphi et C#. Mais JavaScript nécessite un *binding* explicite systématique, comme PHP et quelques autres. C'est pourquoi on ne peut se passer du mot réservé `this`.

Et c'est là que le problème survient : *en JavaScript, `this` peut « se perdre ».*

Il suffit pour cela d'utiliser une référence sur la méthode sans l'appeler directement. Par exemple, pour passer la méthode à un itérateur, ou comme gestionnaire d'événement, ou comme fonction de rappel, ou simplement pour en garder la référence de côté à des fins d'enrobage ou d'appel ultérieur.

Dans un tel cas, on *perd le binding*. Et lorsque la méthode sera appelée, `this` référencera le contexte par défaut, c'est-à-dire, dans un navigateur, l'objet `window`.

Peut-être n'avez-vous pas l'habitude ou la possibilité, dans vos langages de prédilection, d'utiliser des références sur méthodes (parfois appelés « pointeurs sur méthodes » ou « pointeurs sur fonctions membres »). Voici quelques exemples de code qui utilisent tous une référence à la méthode `doSomething` de l'objet `myObj` :

Listing 2-16 Exemples de lignes de code utilisant une référence sur méthode

```
var method = myObj.doSomething;
var toggler = myObj[early ? 'doSomething' : 'doMoreStuff'];
data.each(myObj.doSomething);
window.addEventListener('unload', myObj.doSomething, false);
```

Remarquez qu'on n'a jamais mis de parenthèses après la référence de la méthode : elle n'est donc pas *appelée*, mais juste *référencée*. L'appel se fera plus tard, au gré du code (par exemple, à chaque tour d'un itérateur ou lorsqu'un événement approprié surviendra).

Si la méthode `doSomething` utilise d'autres membres de l'objet `myObj` sur lequel on l'a référencée, elle aura besoin de `this` ; mais ce dernier ne référencera pas `myObj`, il référencera l'objet global `window`. En voici la preuve :

Listing 2-17 La perte du binding en action

```
function MyClass(name) {
    this.name = name;
}
MyClass.prototype.doSomething = function() {
    return 'Hello, my name is ' + this.name + '.';
}
var myObj = new MyClass('Christophe');
var fx = myObj.doSomething;
fx();
// => 'Hello, my name is .'
```

Et voilà, c'est le drame, on a perdu notre nom... En fait, en perdant le *binding* avant d'appeler `fx`, on l'appelle en réalité sur l'objet global `window`, qui a effectivement une propriété `name` (comme toute fenêtre de navigateur), mais dont la valeur par défaut est vide...

C'est évidemment un problème ; comment le résoudre ? JavaScript nous fournit deux outils pour y parvenir, en équipant les fonctions de deux méthodes dédiées : `call` et `apply`. Les deux permettent de préciser l'objet contexte, c'est-à-dire le *binding*, à utiliser pour appeler la méthode concernée. Mais la première prend une liste explicite d'arguments, tandis que la seconde les passe en tant que tableau (ce qui la rend plus pratique pour manipuler des arguments dont on ne connaît pas la nature exacte, d'ailleurs).

Cette possibilité est assez ahurissante par moments, puisqu'elle nous permet d'appeler une méthode appartenant à un type A sur des objets d'un type B qui n'a potentiellement *aucun rapport avec A* ! On voit bien là tout le dynamisme de JavaScript : du moment que l'objet contexte a des propriétés et des méthodes avec les bons noms, ça passera.

Voici un exemple : les tableaux en JavaScript sont dotés d'une méthode `join` qui permet de concaténer les éléments du tableau dans une chaîne de caractères, en les séparant par un délimiteur de notre choix. On aimeraient faire une fonction qui affiche ainsi tous les arguments qui lui sont passés, et comme les fonctions ont toutes une

propriété implicite `arguments`, qui semble se comporter comme un tableau, on serait tenté d'essayer :

Listing 2-18 Tentative d'appel d'une méthode de tableau sur `arguments`

```
['bonjour', 'monde', '!'].join(' ')
// => 'bonjour monde !'

function showArgs() {
  return arguments.join(' ');
}
showArgs('bonjour', 'monde', '!')
// => Erreur, ex. "TypeError: arguments.join is not a function"
```

Qu'à cela ne tienne, rien ne nous empêche d'appeler la méthode `join` du type `Array` sur `arguments`, puisque `join` est *générique* : elle a juste besoin que l'objet sur laquelle on l'appelle *se comporte comme un tableau*. En somme, qu'il ait une propriété `length` et des propriétés numériques allant de 0 à (`length - 1`). Ce type de comportement est parfois appelé *duck typing*. Allons-y :

Listing 2-19 Binding explicite d'une méthode vers un objet quelconque

```
function showArgs() {
  return Array.prototype.join.call(arguments, ' ');
}
showArgs('bonjour', 'monde', '!')
// => 'bonjour monde !'
```

En appelant la méthode `join` du type `Array` avec `arguments` comme contexte explicite (et ' ' comme unique argument), c'est passé.

Évidemment, ça ne résoud pas, tel quel, le problème : il faut ici manifestement avoir la méthode *et* l'objet contexte sous la main au moment de l'appel. On verra au chapitre 4 comment Prototype combine `apply` (une variante plus souple de `call`) et les fermetures lexicales de JavaScript pour fournir une solution élégante et pratique.

Idiomes intéressants

On ne programme bien dans un langage que lorsqu'on commence à rédiger du code idiomatique, c'est-à-dire conforme aux usages du langage. À vouloir écrire du Java comme on faisait du C++, on ne produit que du mauvais Java. À écrire du Ruby sans se défaire de ses habitudes (qu'elles viennent de Perl, Java, C# ou Python), on écrit du mauvais Ruby. Il faut prendre le nouveau langage à bras le corps, et épouser ses concepts, ses formes et ses usages, pour produire quelque chose d'efficace, d'élégant et d'expressif.

JavaScript a traditionnellement été traité de manière assez superficielle, comme un langage de seconde zone dont les utilisations étaient trop simples pour mériter des idiomés de qualité. Et pourtant, dans les bibliothèques qui tirent vraiment partie de la puissance du langage, on trouve de nombreuses perles... Voici quelques bonnes idées extraites du code de bibliothèques comme Prototype ou script.aculo.us.

Initialisation et valeur par défaut avec ||

Commençons par voir comment fournir une valeur par défaut à un argument. On l'a vu, dans la mesure où il est parfaitement possible d'invoquer une fonction avec moins d'arguments qu'elle n'a de paramètres déclarés, certains paramètres vont se retrouver sans valeur.

Si ces paramètres ont ne serait-ce qu'une valeur valide qui soit compatible avec `false` (pour rappel, cela signifie `undefined`, `false`, `-0`, `+0`, `Nan` ou la chaîne de caractères vide, `''`), alors on n'a d'autre choix que de recourir à un test traditionnel, un rien verbeux :

```
if (undefined === arg)
    arg = expressionDeLaValeurParDéfaut;
```

En revanche, quand l'argument n'a comme valeurs valides que des expressions non compatibles avec `false` (un nombre positif, un élément du DOM ou son ID, etc.), alors on peut tirer parti de cette incompatibilité en utilisant la sémantique de court-circuit de l'opérateur logique `||` :

```
arg = arg || expressionDeLaValeurParDéfaut
```

Cet opérateur fonctionne comme en Java, C, C++, C# ou Ruby, pour ne citer qu'eux : si son opérande gauche est équivalent à `true`, il le retourne, sinon il retourne celui de droite. Ici, si notre argument a une valeur valide, on la concerne, sinon on prend celle par défaut.

Si on n'utilise `arg` seulement à un endroit de la fonction, il est même possible de se passer de le redéfinir, et d'employer l'expression directement :

```
function doSomething(requiredArg, arg) {
    ...
    doSubTask(42, arg || expressionDeLaValeurParDéfaut, 'blah');
    ...
}
```

Sélectionner une propriété (donc une méthode) sur condition

Lorsqu'on souhaite choisir parmi deux propriétés d'un objet en fonction d'une condition simple, on a souvent recours, par souci de simplicité, à l'opérateur ternaire `?:`, comme ceci :

```
| var value = someCondition ? obj.propA : obj.propB;
```

Dans tous les langages ayant des fonctions de premier ordre, les fonctions sont, par définition, manipulables comme des valeurs. On peut donc aussi s'en servir pour choisir une méthode :

```
| var fx = someCondition ? obj.methodA : obj.methodB;  
| fx(arg1, arg2);
```

En JavaScript, on peut encore raccourcir cela en profitant du fait que l'opérateur `[]` est, comme on l'a vu, équivalent à l'opérateur de sélection de membre `(.)`, basé sur le nom de la propriété. Par exemple, imaginons un objet ayant deux méthodes d'invocation identiques (prenant toutes deux les mêmes arguments), et qu'on souhaite appeler l'une ou l'autre suivant une condition. On peut simplement écrire :

```
| obj[someCondition ? 'methodA' : 'methodB'](arg1, arg2);
```

Exemple concret dans Prototype, tiré de `Element.toggle` :

```
| Element[Element.visible(element) ? 'hide' : 'show'](element);
```

Encore mieux : appeler une méthode dont le nom est stocké, dynamiquement, dans une variable, mais dont on connaît les paramètres. Là aussi, voici un exemple concret tiré de Prototype (`Form.Element.getValue`) :

```
| var parameter = Form.Element.Serializers[method](element);
```

Personnellement, je trouve cela franchement mignon...

Tester l'absence d'une propriété dans un objet

Nous avons déjà évoqué ce point plus haut avec l'opérateur `in`. Tester qu'un objet ne dispose pas d'une propriété (ce qui inclut les méthodes) ne peut pas toujours se résumer à ceci :

```
| if (obj.propName)
```

Il suffit que l'objet dispose d'une propriété ainsi nommée, dont la valeur est équivalente à `false`, pour que le test ne serve à rien. Préférez l'opérateur `in`, qui sert expressément à ceci :

```
| if ('propName' in obj)
```

Attention à bien encadrer le nom de la propriété par des guillemets simples ou doubles : l'opérateur gauche de `in` est un nom, donc un texte.

Fonctions anonymes : jamais `new Function` !

Nous avons déjà largement eu l'occasion de faire appel aux fonctions anonymes dans ce chapitre, par exemple dans la section sur l'extension des objets par leur prototype, ou sur le *binding*. Je souhaite juste préciser ici qu'il vaut mieux vous abstenir d'utiliser `new Function` dans ce genre de cas, et d'ailleurs, dans tous les cas.

Une fonction anonyme ainsi créée n'a pas le même *binding* pour `this` : au lieu d'utiliser le `this` du contexte environnant, elle utilise le `this` global, donc l'objet `window`. C'est très déroutant. Évitez cette syntaxe, préférez l'opérateur `function` qu'on utilise habituellement. D'autant plus que c'est plus court à écrire !

Objets anonymes comme hashes d'options

Nous avons eu l'occasion de voir des objets anonymes dans nos exemples de *binding*. Un objet anonyme est constitué d'une série de propriétés séparées par des virgules, le tout entre accolades. Chaque propriété (ce qui, je ne le répéterai jamais assez, inclut les méthodes) est définie par son nom, un deux-points (`:`) et sa valeur. Voici un exemple :

Listing 2-20 Un exemple d'objet anonyme

```
var author = {
    name: 'Christophe',
    age: 28,
    publisher: 'Eyrolles',

    greet: function(who) {
        alert('Bonjour ' + who + '.');
    },

    nickname: 'TDD'
}
```

On peut accéder aux propriétés avec l'opérateur point (.). Dans le cas où le nom de la propriété enfreint la syntaxe JavaScript (par exemple, s'il s'agit d'un nombre), on utilisera l'opérateur []. Un objet anonyme se comporte finalement comme un tableau associatif de valeurs, ce qu'on appelle communément un *hash*.

Cette similitude est très pratique lorsqu'on souhaite définir une fonction acceptant un certain nombre d'options, tout en évitant les signatures à rallonge. Une option, par définition, est optionnelle. Représenter une dizaine d'options sous forme d'arguments exigerait de passer toutes les options dans le bon ordre, etc. Ce qui n'est absolument pas agréable.

C'est pourquoi on accepte souvent les options sous forme d'un objet anonyme, dont les propriétés sont nommées d'après les options. Imaginons une fonction avec deux arguments obligatoires, offrant trois options optA, optB et optC ayant pour valeurs par défaut respectivement 42, la chaîne vide ('') et false. On suppose que pour optA, zéro n'est pas une valeur valide. Voici comment réaliser cela facilement :

Listing 2-21 Une fonction avec des options passées comme un hash

```
function myFunc(arg1, arg2, options) {
    // Dans le cas où on ne passe aucune option !
    options = options || {};
    options.optA = options.optA || 42;
    options.optB = options.optB || '';
    options.optC = options.optC || false;
    // Code de la fonction
} // myFunc
```

Évidemment, le code serait encore plus simple si JavaScript proposait un opérateur d'affectation combinée ||=, mais ce n'est pas le cas. Pour appeler notre fonction, on dispose alors d'une syntaxe plutôt agréable, qui n'est pas sans rappeler les arguments nommés. Voici quelques exemples d'appel :

```
myFunc(1, 2);
myFunc(3, 4, { optB: 'Douglas Adams' });
myFunc(5, 6, { optA: 21, optC: true });
```

Simuler des espaces de noms

Lorsqu'on développe une grosse base de code, il est fréquent de vouloir la structurer. Suivant les langages, on dispose de mécanismes appelés paquets, modules, espaces de noms... En JavaScript, on n'a rien de tel (ce qui changera en JavaScript 2, soit dit en passant).

On peut toutefois simuler les espaces de noms en créant des objets anonymes ne jouant qu'un rôle de module. C'est tout simple :

```
var Ajax = {}
Ajax.Base = {
    // Définition de l'objet
}
Ajax.Updater = {
    // Définition de l'objet
}
// etc.
```

« Unobtrusive JavaScript » : bien associer code JS et page web

Il existe de multiples manières de faire en sorte que du code JavaScript soit exécuté dans une page. Du code peut s'exécuter au chargement du script (qui se produit bien avant que la page ne soit complètement chargée), par exemple pour initialiser une bibliothèque, ou en réponse à des événements.

Lorsqu'il s'agit d'associer du JavaScript à des événements, il est impératif que cela ne soit pas fait par le HTML. On ne met pas de JavaScript dans le fichier HTML pour les mêmes raisons qu'on n'y met pas de CSS :

- 1 Cela alourdit chaque page, et fait obstacle aux stratégies de cache des proxies et des navigateurs.
- 2 Cela constitue une intrusion du comportement (ou pour les CSS, de l'aspect) dans le contenu.
- 3 Les possibilités offertes par les attributs HTML sont beaucoup moins riches que celles offertes par le DOM pour l'association de fonctions JavaScript à des événements.

On évitera donc tout script intégré, du style de ceci :

```
<script type="text/javascript">
// du code ici
</script>
```

On évitera aussi tout attribut HTML événementiel, du style `<body onload="...>`. Le seul moyen acceptable de lier du JavaScript à une page consiste à placer ce JavaScript dans un fichier (généralement d'extension `.js`), et à charger ce fichier depuis la section `head` du HTML, comme ceci :

```
| <head>
| ...
|   <script type="text/javascript" src="chemin/nom.js"></script>
| ...
| </head>
```

Au passage, un point important sur l'ancienne syntaxe utilisée pour préciser le langage du script :

```
| <script language="javascript"...
```

Elle est dépréciée depuis HTML 4.01, donc depuis 1999, ce qui remonte déjà à plusieurs années. On utilise désormais l'attribut `type` avec le type MIME du langage, généralement `text/javascript`.

Idéalement, une page doit fonctionner sans aucun JavaScript. Les scripts doivent ajouter du confort, de la facilité, mais ne doivent pas constituer une condition *sine qua non* pour le fonctionnement des pages. Cela irait à l'encontre de l'accessibilité, et serait donc susceptible de gêner de très larges catégories d'utilisateurs.

Parce que la gestion portable, efficace et discrète (*unobtrusive*) des événements – et donc leur association à du code JavaScript – repose sur le DOM, vous verrez le détail de ces possibilités au chapitre suivant, dans la section « Répondre aux événements ». C'est aussi là que vous trouverez un traitement plus détaillé des relations entre JavaScript et accessibilité.

Je signale toutefois un cas où il est envisageable d'avoir une source XHTML avec un fragment de JavaScript à l'intérieur : la récupération d'un fragment XHTML par une requête Ajax. En effet, la couche serveur peut vouloir renvoyer aussi bien un contenu qu'une série d'opérations à effectuer côté client. Nous verrons des exemples de cela au chapitre 7, avec `Ajax.Updater` et son option `evalScripts`.

On peut parfois s'interroger sur le moment d'exécution d'un fragment de script : au moment de son chargement, ou une fois le chargement de la page terminé ? Vous trouverez un traitement détaillé de cette question au chapitre 3, à la section « Ne scripter qu'après que le DOM voulu est construit ».

Astuces pour l'écriture du code

En JavaScript, comme dans beaucoup de langages, on a intérêt à prendre quelques petites habitudes dans la rédaction de notre code source afin d'en améliorer la lisibilité et d'éviter certains pièges classiques. Voici quelques bonnes habitudes que je vous conseille.

Déjouer les pièges classiques

Connaissez-vous les termes *lvalue* et *rvalue*? Ils désignent les deux opérandes d'une affectation : celui de gauche, qui constitue donc généralement une variable à laquelle on peut affecter une valeur, par exemple `index` ou `items[3].nodeValue`; et celui de droite, qui constitue la valeur. Ce dernier est le plus souvent une expression ne constituante pas une variable, par exemple `42, 3 * 7` ou `'hello' + name`.

Un piège fréquent dans de nombreux langages, dont JavaScript, consiste à faire une faute de frappe lors d'une comparaison de type `==`. On oublie un signe d'égalité, et on obtient `=`, l'affectation. Au lieu de ceci :

```
| if (x == 42)
```

on a alors cela :

```
| if (x = 42)
```

Un tel code, qui est très suspect, ne déclenche qu'un avertissement en JavaScript. De tels avertissements ne sont pas immédiatement visibles sur la plupart des navigateurs. Même pour un développeur web équipé d'extensions dédiées, comme la Web Developer Toolbar de Chris Pederick, ou Firebug de Joe Hewitt, un avertissement cause seulement l'apparition d'une petite icône de panneau triangulaire jaune dans un recoin de l'interface : on ne s'en rend pas forcément compte.

Ce code affecte `42` à `x`, et renvoie `x`, qui vaut donc `42`, ce qui est équivalent à `true`, et fait passer la condition. Avec une valeur fixe comme ceci, il est clair que c'est une erreur : si l'on savait qu'on y mettait `42`, on saurait que la condition est fatallement vraie, on n'aurait donc pas mis de `if`.

Pour éviter ce genre de problèmes, je vous recommande de toujours placer les *rvalues* à gauche dans une comparaison `==`. Prenons le code valide suivant :

```
| if (42 == x)
```

Si vous oubliez par mégarde un signe égal, le code obtenu est le suivant :

```
| if (42 = x)
```

Il ne fonctionne pas, tout simplement. Il génère une erreur, qui est souvent plus visible qu'un avertissement, et possède en outre l'immense avantage de stopper l'exécution du script, l'empêchant ainsi de continuer sur la base d'informations fausses, ce qui pourrait poser des problèmes. Imaginez qu'un tel code contrôle l'envoi d'un formulaire de suppression d'utilisateurs, et que le test vérifie qu'une case de confirmation est cochée... On causera certainement des dégâts si on opère sans de telles précautions.

Deuxième conseil : vous aurez peut-être remarqué que JavaScript n'exige pas que vous terminiez systématiquement vos instructions par un point-virgule (;). Il s'agit en effet d'un séparateur d'instructions, et non d'un terminateur ; il est par ailleurs optionnel après les accolades fermantes. Hormis ce dernier cas, je vous conseille néanmoins de toujours utiliser le point-virgule à la fin d'une ligne de code : vous n'aurez pas à vous rappeler de le remettre le jour où vous rajouterez du code à la suite.

Dernier conseil, particulièrement destiné aux débutants : méfiez-vous d'un point-virgule tapé trop hâtivement derrière la parenthèse fermante d'un `if`, `for` ou `while` ! Par exemple, le code suivant :

Listing 2-22 Un script avec de gros problèmes...

```
if (42 < 21);
    alert('youpi !');
var msg = '';
for (var index = 0; index < 10; ++index);
    msg += index + ', ';
while (true);
    if (confirm('Quitter ?'))
        break;
```

Ce code affiche « youpi ! », laisse `msg` à la chaîne vide, et cause une boucle sans fin qui aura tôt fait d'épuiser l'interpréteur. Pourquoi ? Parce que le `if`, le `for` et le `while` ont un point-virgule en trop, discret, en fin de ligne, qui constitue une instruction vide. Comme, syntaxiquement, ces trois structures de contrôle ne gèrent qu'une instruction, c'est cette dernière vide qui est utilisée. En somme, le code précédent est équivalent au code suivant, où l'on repère bien mieux le problème :

Listing 2-23 Le même, reformaté

```
if (42 < 21)
;
alert('youpi !');
var msg = '';
for (var index = 0; index < 10; ++index)
; 
```

```
msg += index + ', ';  
while (true)  
;  
if (confirm('Quitter ?'))  
break;
```

Attention donc à ne pas être trop zélé avec les points-virgules...

Ces conseils valent pour tous les langages dont la syntaxe est proche, voire identique, sur ce point : C, C++, Java, C#...

Améliorer la lisibilité

Je recommande de toujours commenter la fin d'une fonction, au moins lorsque celle-ci dépasse la dizaine de lignes. On se retrouve très vite avec juste la fin d'une fonction visible, que ce soit parce qu'elle est très longue (ce qui indique un problème de modularité du code, soit dit en passant), ou parce qu'on a fait défiler le code source et que seule la fin subsiste. Dans les exemples de ce livre, la plupart des fonctions non triviales ont ainsi un commentaire de fin :

```
function compute(factor) {  
    ...  
} // compute
```

Après une condition ou une définition de boucle, allez systématiquement à la ligne et indentez, c'est tellement plus facile alors de voir quel code dépend de votre structure de contrôle. Évitez ce genre d'indentations impropre :

Listing 2-24 Une indentation très discutable

```
function badCode() {  
    if (arguments.length == 0) throw new Error('Ooops');  
    console.log('some log');  
    for (var index = 0; index < 10; ++index) alert(index);  
    console.info('some other info');  
    while (document.body.hasChildNodes()) clearFirstNode();  
    console.info('done.');
```

Préférez une indentation correcte :

Listing 2-25 Une indentation bien plus utile

```
function betterCode() {  
    if (0 == arguments.length == 0)  
        throw new Error('Missing arguments! Expected name.');//  
    console.log('some log');//  
    for (var index = 0; index < 10; ++index)  
        alert(index);  
    console.info('some other info');//  
    while (document.body.hasChildNodes())  
        clearFirstNode();  
    console.info('done.');//  
} // betterCode
```

Enfin, je ne saurai trop vous encourager à toujours adopter des conventions de nommage dans vos codes source, de préférence alignées sur les standards de l'industrie, ou en tout cas du langage employé. Voici quelques règles de conduite simples, souvent adoptées dans d'autres langages répandus :

- Les types et espaces de noms utilisent une casse dite *CamelCase* majuscule, par exemple Ajax, Uploader ou PeriodicalUpdater.
- Les constantes utilisent une casse majuscule où les composants sont séparés par des traits de soulignement (_): MAX, REGEX_SCRIPTS.
- Les variables, arguments et propriétés utilisent une casse dite *CamelCase* minuscule, c'est-à-dire que la première initiale est minuscule. Quelques exemples d'expressions tout à fait réelles :
`document.documentElement.firstChild.nodeValue, document.createElement.`
- Lorsqu'on crée des objets, je recommande chaudement d'utiliser une convention pour nommer les propriétés censées être privées (non accessibles depuis l'extérieur de l'objet) : préfixez-les par un *underscore* (trait de soulignement), par exemple _id, _each, _observeAndCache(). L'idéal serait d'utiliser un *module pattern* pour les rendre *réellement* privés, mais c'est du JavaScript assez avancé...
- Les noms de méthodes devraient toujours commencer par un verbe à l'infinitif, par exemple createElement, removeChild ou adoptNode. Les méthodes renvoyant un booléen devraient utiliser un verbe à la troisième personne du singulier, généralement au présent de l'indicatif, adapté à la sémantique de la méthode. Exemples : hasChildNodes(), isBlank(), canClose().

Rien qu'en observant ces quelques règles simples, on améliorera sensiblement le code.

Pour aller plus loin

Livres

JavaScript : The Definitive Guide (5th Edition)

David Flanagan

O'Reilly Media, août 2006, 1 018 pages

ISBN 0-596-10199-6

La référence historique, avec tous les détails du langage, et une pléthore d'exemples.
Existe aussi en français dans sa 4^e édition :

JavaScript (4^e édition)

David Flanagan

O'Reilly, septembre 2002, 955 pages

ISBN 2-841-77212-8

JavaScript Bible (5th Edition)

Brendan Eich

Wiley Publishing, mars 2004, 1 236 pages

ISBN 0-764-55743-2

L'autre référence, qui possède l'avantage d'être écrite par l'inventeur du langage, lequel continue de piloter son évolution. En revanche, ne dispose a priori pas d'une version française...

PPK on JavaScript

Peter-Paul Koch

New Riders Publishing, août 2006, 400 pages

ISBN 0-321-42330-5

Un ouvrage tout récent et bien fait par un des principaux gourous du langage. Décrit notamment en détail les interactions entre JavaScript et CSS.

Les expressions régulières par l'exemple

Vincent Fourmond

Eyrolles, août 2005, 126 pages

ISBN 2-914010-65-6

Une bonne façon de se faire les dents sur ce sujet qui fait injustement peur, et dont les bénéfices sont pourtant considérables !

Sites

- La *Mozilla Developer Connection* regorge de ressources précieuses sur JavaScript et le DOM. Je vous conseille en particulier leur guide et leur référence de JavaScript 1.5. Une partie seulement a été traduite en français.
 - <http://developer.mozilla.org/en/docs/JavaScript>
 - <http://developer.mozilla.org/fr/docs/JavaScript>
- Rien ne vaut la spécification officielle pour vérifier un point de détail, ou déterminer si MSIE est fautif ou non. Mais attention, c'est... aride :
<http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- Brendan Eich est l'inventeur et le chef de projet actuel du langage. Son blog permet de suivre l'actualité de JavaScript et de goûter aux futures améliorations :
<http://weblogs.mozilla.org/roadmap/>
- Le site personnel et professionnel de Peter-Paul Koch, dont le livre est cité plus haut : <http://www.quirksmode.org>

3

Manipuler dynamiquement la page avec le DOM

JavaScript est au développeur web ce que les outils sont à l'artisan : le moyen de travailler. Encore faut-il avoir quelque chose sur quoi travailler. Pour un développeur web qui cherche à rendre sa page vivante, cette matière, c'est le DOM, le *Document Object Model*, un ensemble d'interfaces permettant aux langages de programmation d'accéder aux objets qui composent le document. C'est *via* ces interfaces qu'on peut manipuler efficacement la page.

Dans ce chapitre, nous allons voir qu'il n'y a pas *un* DOM mais plusieurs, classés par niveau et par thème. La clef du succès résidant dans la connaissance des quelques interfaces fondamentales, nous consacrerons quelques pages à découvrir celles-ci, en nous concentrant sur les aspects noyau et HTML. Fidèles à l'esprit de l'ouvrage, nous soulignerons quelques pratiques recommandées, avant d'explorer dans le détail la gestion des événements du document. Pour finir, nous construirons quelques exemples concrets très souvent utiles.

Pourquoi faut-il maîtriser le DOM ?

Il est aisé de croire qu'on peut se passer d'étudier les détails du DOM, en particulier lorsqu'on a l'intention de recourir à des bibliothèques ou frameworks tels que Prototype, script.aculo.us ou encore Dōjō. C'est une attitude dangereuse, qui peut entraîner bien des heures perdues au débogage, simplement parce qu'on aura voulu faire l'économie de quelques heures en amont.

La pierre angulaire des pages vivantes

On l'a vu en introduction, il n'y a pas de pages vivantes sans le DOM. Ceux qui en sont encore à manipuler leurs pages « à l'ancienne », ce qu'on appelle aujourd'hui le DOM niveau 0, sont condamnés au recyclage : ces possibilités ne font l'objet d'aucun standard et sont déjà, pour ainsi dire, en préretraite. Gardez-vous de subir le même sort en vous accrochant à des techniques obsolètes !

Ce serait d'autant plus dommage qu'en soi, le DOM est assez simple. Malgré une certaine verbosité et une pléthore d'interfaces, la complexité n'est qu'apparente, et nous avons pour une fois la chance d'avoir un standard plutôt cohérent dans l'ensemble et bien pris en charge par les principaux navigateurs. Une fois certaines règles et certains concepts assimilés, l'utilisation est sans surprise (à l'exception, comme toujours, de certains comportements sur MSIE, que nous aborderons spécifiquement).

Maîtriser la base pour utiliser les frameworks

Quand bien même vous envisageriez d'utiliser des bibliothèques facilitant grandement votre travail, comme Prototype, voire des frameworks entiers tels que script.aculo.us ou Dōjō, cela ne vous dispensera pas de connaître un minimum le DOM.

De nombreuses fonctions de ces bibliothèques sont documentées en utilisant le vocabulaire du DOM : ID d'élément, noeuds fils, noeud père, collection, noeuds textuels sont autant de concepts DOM qui sont utilisés à tout bout de champ par les bibliothèques de plus haut niveau. Sans maîtriser au moins les bases, vous risquez fort de vous y perdre.

Comprendre les détails pour pouvoir déboguer

Les bibliothèques ne sont d'ailleurs pas exemptes de bogues, loin s'en faut. Même lorsque vous utilisez des fonctionnalités stables, rien ne vous empêche de mal interpréter le sens d'un argument, d'en oublier un autre, ou de ne pas satisfaire correctement les exigences d'une fonction.

Lorsque le problème résultant va faire surface, vous n'aurez d'autre choix que de démarrer une séance de débogage ; mais même avec de bons outils à l'appui (débogueur JavaScript et inspecteur DOM sont des incontournables), vous serez bien en peine de comprendre tant le code JavaScript que l'arbre DOM examinés si vous avez négligé d'étudier un peu DOM dans le détail. Pour quelques heures économisées à tort, vous voilà au pied du mur, face à de nombreuses heures fastidieuses et peut-être quelques cachets d'aspirine.

En somme, je ne saurais trop vous recommander de consacrer un peu de temps à lire la suite de ce chapitre. Vos pages web vous le rendront au centuple.

Le DOM et ses niveaux 1, 2 et 3

Le DOM est un standard aux multiples facettes. Non seulement il évolue dans le temps, au travers de niveaux (qui tiennent lieu de versions), mais en plus il est modularisé par thèmes, ou aspects si vous préférez.

Vue d'ensemble des niveaux

Voici un tour d'horizon des niveaux successifs du DOM. Les spécifications W3C indiquées peuvent être arides à consulter pour le débutant : si vous voulez vous y plonger rapidement, vous gagnerez à lire d'abord les conseils de l'annexe C.

Tableau 3-1 Les niveaux du DOM

Niveau	Description
0	<p>Il ne s'agit pas d'un standard à proprement parler, mais du nom sous lequel on désigne aujourd'hui les possibilités de manipulation du document « pré-DOM », telles qu'on en trouve encore dans de trop nombreux didacticiels et ouvrages (tous datant un peu, il est vrai). Par exemple, le code suivant :</p> <pre>document.forms['inscription'].nom_client.value</pre> <p>...est très clairement du niveau 0 : la collection forms ne propose pas d'opérateur [] dans le DOM, même si sa fonction namedItem a un sens équivalent (et encore, pas en XHTML). Quant à l'utilisation d'un objet fils nommé d'après l'attribut name du champ, elle est totalement absente du DOM également : on préfère accéder directement au champ par son ID, où qu'il soit dans le document.</p> <p>Peter-Paul Koch propose un excellent article sur ce niveau :</p> <p>http://www.quirksmode.org/js/dom0.html.</p>
1	Ce niveau est le premier véritable standard du DOM, remontant à 1998. Il n'était pas encore découpé en modules et fournissait déjà les principales interfaces pour le noyau et HTML. Dernière révision officielle : http://www.w3.org/TR/REC-DOM-Level-1/ .

Tableau 3–1 Les niveaux du DOM (suite)

Niveau	Description
2	<p>L'ensemble des modules du niveau 2 date du 13 novembre 2000, mais le module HTML a subi cinq révisions depuis, la dernière remontant à janvier 2003. Ce niveau ajoute principalement la prise en charge de XHTML 1.0, avec entre autres impacts une déclinaison de nombreuses méthodes pour gérer les espaces de noms. L'interface <code>HTMLOptionsCollection</code> fait son apparition.</p> <p>Le niveau 2 est celui le plus largement pris en charge par les principaux navigateurs actuels. Nous nous intéresserons principalement aux modules noyau, événements et HTML :</p> <ul style="list-style-type: none"> • http://www.w3.org/TR/DOM-Level-2-Core/core.html • http://www.w3.org/TR/DOM-Level-2-Events/events.html • http://www.w3.org/TR/DOM-Level-2-HTML/html.html
3	<p>Le travail actuel porte sur le niveau 3, état de l'art du DOM. Il modifie principalement les aspects noyau, vues/formatage et événements, et ajoute plusieurs modules : Load and Save, Validation et XPath. En regard des possibilités déjà existantes, c'est surtout ce dernier module qui ajoute de la valeur, à mon sens. Nous l'utiliserons d'ailleurs au chapitre 5, lorsque nos requêtes Ajax récupéreront un contenu XML.</p> <p>Ce niveau vise surtout à ajouter des possibilités de l'univers XML à nos pages (comparaison de noeuds et d'arbres, gestion de l'encodage, des URI relatifs, de la normalisation, de la validité vis-à-vis des schémas et DTD), et à simplifier certaines opérations (par exemple grâce à <code>adoptNode</code>, <code>renameNode</code>, <code>textContent</code> et <code>wholeText</code> ou aux mécanismes permettant d'associer des données utilisateur aux noeuds). La gestion des événements est aussi mieux spécifiée.</p> <p>Certains modules sont déjà finalisés : Noyau, Load and Save et Validation. Les autres sont encore en travaux. Ceci dit, les dernières révisions datent d'avril 2004, le W3C faisant là aussi la preuve de son extraordinaire rapidité... On citera principalement le module noyau :</p> <ul style="list-style-type: none"> • http://www.w3.org/TR/DOM-Level-3-Core/core.html

Support au sein des principaux navigateurs

Comme on l'a vu en avant-propos, les principaux navigateurs prennent plutôt bien en charge le DOM dans son niveau 2. C'est le cas de Mozilla, Firefox, Camino, Safari, Konqueror et Opera, pour ne citer qu'eux. MSIE dispose aussi d'une bonne prise en charge, malgré quelques écarts hélas problématiques, notamment pour la gestion des listes dans les formulaires (on y reviendra dans le détail plus tard).

Certains fournissent partiellement le niveau 3. C'est par exemple le cas de Firefox et donc Camino (notamment le module XPath et les possibilités XSLT !), Opera (XPath et Load and Save) et Konqueror. Côté MSIE, il ne faut pas s'y attendre avant au moins la version 8, soit dans plusieurs années...

Les aspects du DOM : HTML, noyau, événements, styles...

Nous l'avons déjà évoqué, le standard DOM est aujourd'hui divisé en modules, ou aspects, afin de mieux structurer son contenu et de faciliter la consultation. En voici un rapide tour d'horizon (les modules marqués d'un astérisque ne disposent pas d'une version finalisée à l'heure où j'écris ces lignes) :

Tableau 3–2 Modules du DOM avec leur niveau d'apparition et leur dernière révision

Module	Apparu dans le niveau	Dernière version	Description
Noyau (Core)	1	07/04/2004	Interfaces fondamentales (voir prochaine section).
HTML	1	09/01/2003	Interfaces spécifiques aux contenus des documents (X)HTML.
Vues (Views)	2	13/11/2000	Manipulation d'une représentation visuelle spécifique d'un document (par exemple une fois qu'une feuille CSS aura été appliquée).
Événements (Events)	2	13/11/2000	Gestion événementielle (écoute, traitement, propagation...).
Style	2	13/11/2000	Manipulation des feuilles de styles et des styles calculés pour chaque nœud.
Traversal and Range	2	13/11/2000	Parcours et filtrage d'un document (<i>traversal</i>) et manipulation de fragments de documents (<i>range</i>).
Load and Save	3	07/04/2004	Stockage et récupération de documents par (dé)sérialisation.
Validation	3	15/12/2003	Vérification de la conformité du document à sa grammaire (DTD/schéma).
XPath*	3	26/02/2004	Extraction de nœuds du document à l'aide de la syntaxe XPath.
Views and Formatting*	3	26/02/2004	Extensions au module Vues.
Abstract Schemas*	3	27/07/2002	Manipulation des grammaires de document (DTD, schéma, etc.).

Maîtriser les concepts : document, nœud, élément, texte et collection

Il faut d'abord comprendre que le DOM au sens W3C s'applique à un document de type XML. Cela n'implique pas forcément que la syntaxe du document soit conforme à XML (même si c'est généralement le cas), mais que le document résultat soit d'une nature identique : un arbre de nœuds imbriqués les uns dans les autres.

Par conséquent, dans le cadre de pages web, on pourra exploiter le DOM d'autant plus efficacement (et avec moins de surprises potentielles) qu'on utilisera un balisage

conforme à XHTML 1 strict. Un balisage plus laxiste, moins cohérent, du type de HTML 4.01, peut engendrer un DOM parfois... inattendu !

Si ces notions de balisage sémantique et de différences entre XHTML et HTML vous semblent un peu confuses, n'hésitez pas à faire un tour à l'annexe A.

Le DOM de votre document : une arborescence d'objets

Un DOM est donc une arborescence d'objets, ou plus généralement de noeuds, qui représente le document. C'est-à-dire, dans le cas qui nous concerne, la page web. On accède au document lui-même au travers d'une interface `Document`. Lorsqu'on part de là pour explorer le contenu du document, on ne tombe que sur des noeuds, mais chacun d'un type spécifique.

Les principaux types de noeuds reflètent les différentes catégories de balisage possibles dans un document XML (donc XHTML) : éléments, attributs, commentaires, textes (ce qu'on met généralement entre la balise ouvrante et la balise fermante), mais aussi des types moins courants, comme « instruction », le type des déclarations `DOCTYPE` par exemple.

Tous ces types partagent un ensemble de propriétés et fonctions héritées de leur type générique : le noeud (`node`). Mais ils ont aussi leurs spécificités, représentées par leurs interfaces dédiées (par exemple, `Element` pour les éléments, ou `Attr` pour les attributs).

L'arborescence du DOM est souvent bien plus verbeuse que le balisage XHTML correspondant, et c'est souvent une source de surprise pour les débutants. Si on n'y prend pas garde, elle peut causer des confusions responsables de bien des bogues dans les scripts exploitant le DOM.

Voici un document XHTML d'exemple :

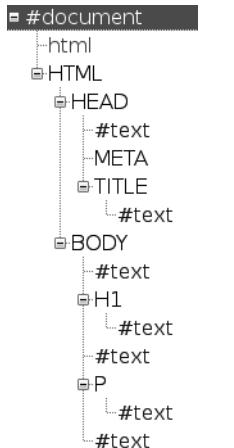
Listing 3-1 Un document XHTML simpliste

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  > xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    > charset=iso-8859-15" />
  <title>Un document tout petit</title>
</head>
<body>
  <h1>Un document tout petit</h1>
  <p>Ceci est un petit document</p>
</body>
</html>
```

Derrière ce document se cache une arborescence DOM plus épaisse qu'on ne pourrait croire !

Figure 3–1

Le DOM de ce document, vu par l'inspecteur DOM de Firefox



On voit bien que la racine est un objet `Document` ; le `html` minuscule représente un nœud de type instruction, qui correspond à la déclaration `DOCTYPE` en haut du code source (`html` est le nom de l'élément racine défini par la déclaration). Mais que sont tous ces nœuds `#text` ?! Et pourquoi les balises sont-elles en majuscules ?

Le plus simple d'abord : les balises apparaissent en majuscules parce que, même si XHTML exige des balises minuscules (car XML est sensible à la casse, et la DTD définit les balises en minuscules), le DOM utilise les noms canoniques des balises, ce qui implique généralement qu'ils soient en majuscules.

La raison d'être des `#text` n'est guère plus compliquée. Comme nous le verrons à plusieurs reprises dans les sections suivantes, consacrées aux principales interfaces du DOM noyau et HTML, une balise non vide (c'est-à-dire une balise disposant de versions ouvrante et fermante, et ayant au moins un caractère entre les deux) ne contient pas directement le texte qui y figure. On a ici deux nœuds : un nœud de type élément pour la balise à proprement parler (parties ouvrante et fermante), et un nœud de type texte pour le texte figurant entre les deux.

Prenons le fragment suivant :

```
<title>Un document tout petit</title>
```

Dans le DOM, ce fragment est représenté par deux nœuds, comme on peut le voir sur la figure 3-1 :

- 1 Un nœud élément nommé TITLE, sans valeur.
- 2 Un nœud texte sans nom (d'où le #text), de valeur Un document tout petit.

Ce que nous prenons pour des espaces sans signification particulière – sous prétexte qu'un navigateur va représenter la plupart des séries d'espaces (et retours chariot, tabulations, etc.) sous forme d'une seule espace à l'affichage – ne disparaît pas pour autant du document, et se retrouve donc dans le DOM. Ainsi, les nœuds texte figurant avant H1, entre H1 et P, et après P, représentent les lignes vides entre ces éléments.

Nous verrons un peu plus tard, en parlant de l'interface Text, que le type MIME du document (principalement text/html ou application/xhtml+xml) influe sur les caractères qu'on trouve dans les nœuds texte ainsi que sur les ajustements que le navigateur peut réaliser à la création du DOM. Par exemple, si vous regardez attentivement la figure 3-1 et la comparez au listing 3-1, vous verrez qu'il manque apparemment un nœud texte entre META et TITLE.

Voyons maintenant une présentation des principales interfaces, en commençant par celles du module noyau.

Node

C'est l'une des deux interfaces incontournables du DOM. Tous les nœuds d'un document, quel que soit leur type spécifique, sont avant tout des nœuds. Chaque interface dédiée hérite donc de l'interface Node. Du coup, toutes les interfaces sur nœuds que vous manipulerez (éléments, attributs, textes...) disposeront des propriétés et méthodes de Node.

On peut donc tout faire rien qu'en utilisant ces méthodes, mais lorsqu'on manipule des éléments, on utilisera généralement les méthodes de plus haut niveau de l'interface Element, notamment pour manipuler les attributs (sinon, c'est l'enfer !). Et dans le cadre précis de pages web, on aura recours à toutes les propriétés prédéfinies pour chaque type de balise.

Voici les quatre propriétés fondamentales de Node.

Tableau 3-3 Propriétés fondamentales de Node

Propriété	Description
nodeName	Le nom du nœud. Il varie suivant le type de nœud. Cas principaux : pour un élément, équivalent de la propriété tagName ; pour un nœud texte, #text ; pour un attribut, équivalent de la propriété name.

Tableau 3–3 Propriétés fondamentales de Node (suite)

Propriété	Description
nodeType	Le type de nœud. Valeur numérique indiquant le type de nœud (élément, texte, attribut, etc.). Les navigateurs offrant une prise en charge propre fournissent les constantes pour l'ensemble des types, tandis que les autres vous laissent vous débrouiller avec les valeurs numériques. Voir plus bas pour les principales valeurs.
nodeValue	Valeur du noeud. Elle varie suivant le nœud. Pour un attribut, c'est l'équivalent de la propriété value ; pour un texte, un commentaire ou une section CDATA, eh bien... c'est le texte ! ; pour la plupart des autres types, ça vaut null.
attributes	On se demande ce que cette propriété fait là plutôt que dans l'interface Element, puisqu'elle n'est définie que pour les nœuds de type élément ! Il s'agit d'un tableau à valeurs nommées (interface NamedNodeMap, voir plus bas) fournissant tous les attributs de l'élément. On ne s'en sert que pour lister les attributs dynamiquement : pour un attribut précis, on utilisera getAttribute et setAttribute, définis dans l'interface Element, et bien plus pratiques.

Les principales constantes pour les types de nœud sont (valeur et nom) :

- 1 Node.ELEMENT_NODE
- 2 Node.ATTRIBUTE_NODE
- 3 Node.TEXT_NODE

Voyons à présent les propriétés qui permettent de se déplacer dans le DOM, tâche absolument critique pour la plupart des scripts.

Tableau 3–4 Propriétés de parcours du DOM de Node

Propriété	Description
parentNode	Le nœud père dans l'arborescence. Il aura pour valeur null si le nœud vient d'être créé ou s'il s'agit du noeud Document (tout en haut de l'arborescence), d'un attribut ou de quelques autres types ésothériques. Dans le cas contraire, indique le nœud parent, ou conteneur, du nœud courant. Par exemple, le nœud père de title est normalement head, celui de head est html, celui de html est #document et la chaîne s'arrête là.
childNodes, firstChild, lastChild	En descente maintenant : ces propriétés permettent d'accéder aux nœuds fils du nœud courant. Un élément vide (<balise ... />) n'aura pas de nœuds fils, pas plus qu'un attribut, un commentaire ou une instruction. Dans un tel cas, childNodes renverra une liste vide (0 == childNodes.length), et firstChild comme lastChild renverront null. Sinon, la liste n'est pas vide, firstChild renvoie le premier nœud fils et lastChild le dernier nœud fils. S'il n'y a qu'un nœud fils, on a donc firstChild == lastChild !
previousSibling, nextSibling	Renvoient les nœuds frères précédent et suivant, respectivement. Un nœud frère est un nœud ayant le même noeud parent que le nœud courant, donc figurant au même niveau dans le document. Les notions de précédent et suivant respectent l'ordre du document.

Pour bien fixer les idées, voici un petit fragment de XHTML et quelques résultats d'expressions. On utilisera dans le code source JavaScript des variables nommées d'après les attributs `id` du source XHTML (ce qui ne se fait pas tout seul : on suppose qu'on a correctement défini ces variables auparavant).

```
<h1 id="header">Personnes inscrites</h1>
<ul id="people">
  <li id="al">Alexis</li>
  <li id="nioute">Anne-Julie</li>
  <li id="elodie">Élodie</li>
  <li id="mimi">Marie-Hélène</li>
  <li id="xavier">Xavier</li>
</ul>
```

On a alors :

```
header.nextSibling.nodeType == Node.TEXT_NODE // Surprise !
header.nextSibling.nextSibling == people
nioute.lastChild == nioute.firstChild
elodie.nextSibling.nextSibling == mimi
people.previousSibling.previousSibling == header
header.childNodes.length == 1
header.firstChild.nodeType == Node.TEXT_NODE
// Ci-dessous : 6 noeuds texte entrelacés à 5 éléments
people.childNodes.length == 11
people.firstChild.nodeType == '#text' // Surprise !
people.firstChild.nodeValue == "\n    " // En mode HTML en tout cas
elodie.firstChild.nodeName == '#text'
xavier.lastChild.nodeValue == 'Xavier'
xavier.parentNode.firstChild.nextSibling == al
// Ci-dessous, valide en mode HTML. En mode XML, serait 'h1'
nioute.parentNode.nodeName == 'UL'
```

On a souvent tendance à oublier ces maudits nœuds texte un peu partout, dus à l'indentation, aux retours à la ligne... Avec un peu d'habitude, on finit par s'y habituer, et arrêter de voir surgir des erreurs JavaScript du type « pas de propriété `xxx` pour le nœud ». Et surtout, avec les méthodes étendues de l'objet `Element` dans Prototype (que nous étudierons au chapitre 4), on se déplace beaucoup plus simplement et efficacement !

Certains écrivent rapidement des fonctions du style `firstElementChild`, `lastElementChild`, `previousElementSibling` et `nextElementSibling`, pour éviter d'avoir à penser à cela, mais il s'agit à mon sens d'une fausse bonne idée. En effet, le simple fait d'utiliser nos fonctions propres plutôt que les fonctions natives du DOM nous amène à penser au problème ! Qui plus est, cela crée une dépendance plutôt superflue à nos quatre petites fonctions, que nous devrons d'ailleurs utiliser comme

des fonctions classiques (`x = firstElementChild(node)`) plutôt que comme des propriétés (`x = node.firstChild`). Le jeu n'en vaut pas la chandelle. Pour les besoins complexes de parcours, on utilisera simplement des mécanismes plus avancés que ces propriétés.

Après avoir passé en revue les propriétés, examinons les méthodes de `Node`, qui sont très fréquemment utilisées, puisqu'elles constituent le seul moyen d'altérer le DOM en ajoutant, déplaçant ou retirant des noeuds.

Tableau 3–5 Principales méthodes de Node

Méthode	Description
<code>appendChild(newChild)</code>	Ajoute le noeud <code>newChild</code> en dernière position à l'intérieur du noeud courant (<code>newChild</code> devient automatiquement le <code>lastChild</code>). Astuce : si <code>newChild</code> était déjà présent dans le DOM, il est automatiquement retiré d'abord. Un déplacement se fait donc en un seul appel de méthode.
<code>cloneNode(deep)</code>	Crée un clone du noeud courant, sans l'attacher au DOM. Le paramètre <code>deep</code> est un booléen indiquant s'il s'agit d'une copie profonde (incluant tous les noeuds fils, donc une copie de l'arbre dont le noeud courant est racine) ou d'une copie superficielle (juste le noeud courant, sans ses noeuds fils). Ce dernier cas est très rare, on fait donc généralement un <code>cloneNode(true)</code> . En somme, c'est un peu la même chose qu'un constructeur copie.
<code>hasAttributes()</code>	Indique si le noeud (qui doit être un élément) a des attributs. Méthode introduite au niveau 2, et cohérente avec la présence de la collection <code>attributes</code> . Équivalent propre de <code>0 != attributes.length</code> .
<code>hasChildNodes()</code>	Permet de savoir si le noeud a des noeuds fils. Équivalent propre de <code>0 != childNodes.length</code> .
<code>insertBefore(new, ref)</code>	Insère un noeud fils à une position bien déterminée. Le noeud à insérer est passé en premier (<code>new</code>), le noeud de référence en second. Comme le nom de la méthode l'indique, <code>new</code> sera positionné avant <code>ref</code> . Ainsi, pour insérer <code>n</code> comme premier fils de <code>p</code> : <code>p.insertBefore(n, p.firstChild)</code> . Notez que l'insertion peut avoir lieu même quand <code>p</code> n'a pas encore de noeud fils, car <code>p.insertBefore(n, null)</code> est équivalent à <code>p.appendChild(n)</code> . Comme pour <code>appendChild</code> , si <code>new</code> était déjà présent dans le DOM, il est d'abord retiré automatiquement.
<code>removeChild(old)</code>	Retire le noeud fils <code>old</code> du DOM. C'est le seul moyen de retirer un noeud : on n'a pas de méthode <code>remove</code> ou <code>delete</code> qui supprime le noeud sur lequel elle est invoquée : il faut demander à son noeud parent (attention à ne pas passer <code>null</code>). Notez bien que je dis retirer, pas détruire. Le noeud existe toujours en mémoire, vous pouvez garder une référence dessus. Je rappelle que JavaScript n'a pas de destructeurs d'objets, c'est un langage à « ramasse-miettes » (garbage collector).
<code>replaceChild(new, old)</code>	Remplace le noeud fils <code>old</code> par le noeud <code>new</code> . Renvoie <code>old</code> . Comme pour <code>appendChild</code> et <code>insertBefore</code> , si <code>new</code> était déjà présent dans le DOM, il est d'abord retiré automatiquement.

Petite précision (évidente à l'utilisation mais sait-on jamais) : tous les ajouts, retraits et remplacements concernent le nœud (`new` ou `old`) et ses nœuds fils, c'est-à-dire le nœud en tant que fragment de document.

Avant de pouvoir manipuler ces méthodes dans des exemples, nous allons devoir explorer l'interface `Document`, qui nous permettra de mettre la main sur des éléments existants (pour les déplacer, les modifier ou les retirer du DOM), et d'en créer de nouveaux pour les ajouter au DOM.

Notons enfin que le DOM niveau 3 a rajouté quelques méthodes et propriétés de confort à l'interface `Node`, en particulier `isSameNode`, `isEqualNode`, `getUserData`, `setUserData` et `textContent`. Il s'agit ici surtout de raccourcir le code nécessaire à quelques tâches courantes.

Document

C'est l'autre interface incontournable, quoi que vous vouliez faire avec le DOM. C'est grâce à elle qu'on accède au document, comme son nom l'indique. Elle est la plupart du temps fournie par un objet JavaScript appelé `document`. Cet objet a d'ailleurs la bonne idée d'implémenter aussi l'interface `DOMImplementation`, que nous verrons plus loin. En somme, tout fragment de script DOM utilise `document`.

Voici les principales propriétés et méthodes.

Tableau 3–6 Principales propriétés et méthodes de Document

Propriété/méthode	Description
<code>documentElement</code>	L'élément (interface <code>Element</code>) racine du document. Dans un document (X)HTML, c'est l'élément <code>html</code> . Pratique pour réaliser un parcours manuel en partant de la racine.
<code>createElement(tagName)</code>	Le seul et unique moyen de créer un élément en JavaScript, donc d'enrichir ensuite le document. On passe le nom de la balise en argument. Je conseille les minuscules, qui fonctionneront aussi sur un document 100 % XML, alors que la forme canonique (majuscule) échouerait. Renvoie le nœud fraîchement créé (interface <code>Element</code>). Attention : de nombreux exemples en ligne passent un fragment HTML complet, du style <code>document.createElement('<div id="test">Bonjour</div>')</code> . C'est une extension partielle de MSIE, ce qui ne fait absolument pas partie du standard.
<code>createTextNode(data)</code>	Le seul moyen de créer un nœud texte, l'autre grand type de nœud dans une page web. On passe le texte du nœud en argument. Renvoie le nœud texte (interface <code>Text</code>) ainsi construit.

Tableau 3–6 Principales propriétés et méthodes de Document (suite)

Propriété/méthode	Description
getElementsByName(tag)	Très pratique, permet de récupérer une liste (interface <code>NodeList</code>) de tous les éléments, dans l'ordre du document (parcours en profondeur, de haut en bas), dont le nom a été passé en argument. Là aussi, on est sensible à la casse lorsqu'on est en mode XML. Par exemple, pour récupérer tous les éléments <code>table</code> d'une page, on utilisera <code>document.getElementsByName('table')</code> . La valeur spéciale <code>*</code> est parfois utile et permet de récupérer tous les éléments du document.
getElementById(id)	Recherche dans tout le document le nœud dont l'ID (attribut <code>id</code>) est fourni. Renvoie l'élément (interface <code>Element</code>) en cas de succès, <code>null</code> sinon. C'est la méthode incontournable. Inutile d'espérer faire un script utile sans elle. Pour beaucoup, ne pas en disposer revient à ne pas disposer du DOM, et par extension, tester qu'on a le DOM revient à tester, en fait, qu'on a au moins le niveau 2.

Deux mots sur les méthodes `createElement`. D'abord, les nœuds sont bien créés en mémoire et nous sont renvoyés pour manipulation ultérieure, mais ils ne figurent pas encore dans le DOM, ils ne sont rattachés à aucune portion existante du document. Il nous appartient de les insérer à l'endroit qui nous intéresse, avec les méthodes de l'interface `Node`. Ensuite, pour `createElement`, si le type d'élément indiqué prévoit des valeurs par défaut pour ses attributs, les nœuds `Attr` correspondants sont automatiquement créés et ajoutés à sa liste `attributes`.

On trouve également les variantes à suffixe `NS` de nombreuses méthodes, qui prennent en charge un paramètre supplémentaire indiquant un espace de noms. On est alors clairement en XHTML, car il s'agit d'une fonction importante héritée du XML, qui permet de faire cohabiter plusieurs vocabulaires dans un même document (par exemple, pour utiliser des balises MathML ou SVG).

Le niveau 3 a ajouté quelques méthodes bien utiles, comme `adoptNode`, `normalizeDocument` et `renameNode`, qui facilitent des tâches assez fréquentes mais auparavant un brin verbeuses à programmer.

Voici à présent quelques exemples, pour se faire une première idée.

Listing 3-2 Un script plutôt indécis qui illustre beaucoup de choses

```
// Création d'un équivalent <h1>Bonjour</h1>
var header = document.createElement('h1');
header.appendChild(document.createTextNode('Bonjour'));

// Ajout au début de <body>
var body = document.getElementsByTagName('body').item(0);
body.insertBefore(header, body.firstChild);
```

```
// Finalement plutôt après l'en-tête d'ID 'main'...
var mainHdr = document.getElementById('main');
mainHdr.parentNode.insertBefore(header,
    mainHdr.nextSibling);

// Et puis carrément, en remplacement de l'en-tête !
mainHdr.parentNode.replaceChild(mainHdr, header);

// D'ailleurs, on ne dit jamais assez bonjour... x2 !
header.parentNode.insertBefore(header.cloneNode(true), header);

// En revanche, ceci donnerait un h1 vide...
header.cloneNode(false);
// ... car on n'a pas cloné le nœud texte fils de header !
```

Attention tout de même à faire les choses plus directement dans votre code de production ! Le script ci-dessus se résumerait à ceci, s'il ne prenait pas tant de détours :

Listing 3-3 La version minimale du script précédent

```
var header = document.createElement('h1');
header.appendChild(document.createTextNode('Bonjour'));

var mainHdr = document.getElementById('main');
mainHdr.parentNode.replaceChild(header, mainHdr);
header.parentNode.insertBefore(header.cloneNode(true), header);
```

Element

L'interface `Element` étend les capacités de `Node` pour les nœuds représentant des éléments, c'est-à-dire des balises. Ces nœuds se distinguent principalement des autres en ce qu'ils peuvent avoir des attributs et des éléments fils¹.

Plutôt que de devoir créer à la main des nœuds `Attr` et les gérer individuellement, en plus de les ajouter ou de les retirer à la liste `attributes` de l'élément, on dispose donc de méthodes dédiées, qui facilitent le travail. On verra plus tard, avec le DOM HTML, que même ces méthodes sont peu utilisées : on passe généralement par des propriétés spécifiques pour chaque attribut concret. Néanmoins, pour pouvoir réaliser un traitement générique, il faut les connaître.

1. D'accord, `Document` peut aussi avoir des éléments fils... Mais si on ne peut plus simplifier à bon escient...

Tableau 3–7 Propriétés et méthodes principales de l'interface Element

Propriété/méthode	Description
tagName	La seule propriété significative. Fournit le nom de la balise pour l'élément. En mode HTML, utilise la forme canonique (majuscule), en mode XML et autres modes sensibles à la casse, utilise la casse employée à la création. La propriété nodeName devient synonyme de celle-ci.
getAttribute(name)	Renvoie la valeur de l'attribut indiqué, ou la chaîne vide (' ') si l'attribut n'existe pas. Prend en compte une éventuelle valeur par défaut.
getElementsByTagName(tag)	Tiens ! On l'avait déjà au niveau de Document, celle-ci. Mais justement : appelée sur un élément, elle restreint son champ de recherche au fragment du document situé à l'intérieur de cet élément. Potentiellement beaucoup plus performant, et souvent plus utile que la version globale.
hasAttribute(name)	Précise si l'attribut indiqué est présent (ou dispose d'une valeur par défaut, ce qui revient au même). Plutôt utile, car getAttribute ne distingue pas entre attribut à valeur vide et attribut absent. Attention : n'est pas pris en charge par MSIE (mais Prototype vous évitera de vous en soucier).
removeAttribute(name)	Retire un attribut à l'élément. Attention : si l'attribut de ce nom dispose d'une valeur par défaut, une nouvelle définition d'attribut avec cette valeur prend la place de la définition retirée. Il n'est donc pas possible de retirer une valeur par défaut. Si l'attribut n'existe pas, n'a aucun effet.

En somme, rien de bien révolutionnaire par rapport à ce que Node permettait de faire, surtout dans la mesure où, dans la majorité des codes rencontrés en pratique, on utilisera des propriétés spécifiques, issues du DOM HTML, pour les attributs qui nous intéressent.

Text

C'est le type des noeuds représentant un fragment de texte. En simplifiant un peu (au mépris de quelques cas particuliers plutôt rares), pour un document HTML, un fragment de texte démarre au premier caractère après une balise, et se termine au premier caractère <. Il comporte donc toute la mise en forme du code source HTML : retours chariot, indentation (par espaces ou tabulations), etc. C'est pourquoi on a tant de noeuds texte « inutiles » au milieu de nos arbres DOM.

Techniquement, l'interface n'hérite pas immédiatement de Node : elle dérive de CharacterData, qui elle-même dérive de Node. On dispose donc des méthodes de CharacterData, qui permettent de modifier la valeur *in situ*, en insérant du texte au milieu de l'existant, en en supprimant ou remplaçant une portion, voire en remplaçant tout.

Ces méthodes, appendData, insertData, deleteData et replaceData, sont assez rarement utilisées, car elles vont au-delà des besoins courants. Même la propriété

officielle pour le texte, `data`, n'est pratiquement jamais utilisée nommément, la propriété générique `nodeValue` étant ici synonyme.

En revanche, on utilise trop peu la propriété `length`, qui contient la taille pré-calculée du contenu texte. On passe trop souvent par la méthode `length()` des objets `String`. Ainsi :

```
header.firstChild.length
```

est équivalente, mais potentiellement plus efficace (au niveau de la milliseconde, entendons-nous bien...) que :

```
header.firstChild.nodeValue.length()
```

Les quelques méthodes restantes, `splitText` et les nouveautés de niveau 3, `wholeText` et `replaceWholeText`, correspondent à des usages pour l'instant si rares qu'on ne fera pas plus que les mentionner.

NodeList et NamedNodeMap

Ce sont les deux principales interfaces pour manipuler des collections, c'est-à-dire, pour simplifier des listes de noeuds. La différence entre les deux est que `NodeList` utilise uniquement des positions (1^{er} élément, 4^e élément...) tandis que `NamedNodeMap` associe un nom à chaque noeud.

`NodeList` est très fréquemment utilisée, car c'est le type de `childNodes` et le type de retour de `getElementsByTagName`. Sa définition est triviale.

Tableau 3-8 Propriété et méthode de l'interface `NodeList`

Propriété/Méthode	Description
<code>item(index)</code>	Accède au noeud (interface <code>Node</code>) concerné. Il peut s'agir de n'importe quel type de noeud, bien entendu. Le premier noeud est en position 0 (zéro), le dernier en position <code>length - 1</code> . Toute position supérieure ou égale à <code>length</code> renverra <code>null</code> .
<code>length</code>	Taille de la liste.

L'interface JavaScript officielle pour `NodeList` permet d'utiliser l'opérateur `[]` comme équivalent de la méthode `item` : `maListe[index]` est donc équivalent à `maListe.item(index)`. En JavaScript, une `NodeList` est donc « compatible tableau ».

`NamedNodeMap` est principalement utilisée comme type de la propriété `attributes`. On ne s'en sert donc que dans le cadre de traitements génériques. Sa définition est des plus simples, elle aussi .

Tableau 3–9 Principales propriétés et méthodes de l'interface NamedNodeMap

Propriété/Méthode	Description
<code>getNamedItem(name)</code>	Renvoie le nœud associé au nom passé, ou <code>null</code> si aucun nœud n'est associé à name.
<code>item(index)</code>	Accède au nœud (interface Node) concerné. Il peut s'agir de n'importe quel type de nœud, bien entendu. Le premier nœud est en position 0 (zéro), le dernier en position <code>length - 1</code> . Toute position supérieure ou égale à <code>length</code> renverra <code>null</code> .
<code>length</code>	Taille de la liste.
<code>removeNamedItem(name)</code>	Retire l'association basée sur le nom passé. Dans la mesure où NamedNodeMap n'est utilisée que pour <code>attributes</code> , le fonctionnement est similaire à celui de <code>removeAttribute</code> : si une valeur par défaut est définie, un nouveau nœud avec cette valeur est créé en remplacement du nœud retiré. Attention toutefois en cas de nom inexistant (pas d'association) : lève une exception <code>NOT_FOUND_ERR</code> .
<code>setNamedItem(node)</code>	Stocke le nœud comme valeur associée à <code>node.nodeName</code> . On évite donc d'y stocker plusieurs nœuds texte, par exemple, puisqu'ils ont tous le même nom (<code>#text</code>). Ceci dit, encore une fois, les seules NamedNodeMaps qu'on emploie en général sont les propriétés <code>attributes</code> et on les utilise principalement en consultation. Si on devait les modifier, on passerait des nœuds <code>Attr</code> , qu'on ne détaille pas ici.

À titre d'exemple, voici une petite fonction qui obtient en quelque sorte le `innerText` d'un nœud (élément ou texte), en concaténant récursivement les valeurs de tous ses nœuds texte internes.

Listing 3–4 Un exemple de parcours du DOM et d'utilisation de NodeList

```
function getInnerText(node) {
    var result = '';
    if (Node.TEXT_NODE == node.nodeType)
        return node.nodeValue;
    if (Node.ELEMENT_NODE != node.nodeType)
        return '';
    for (var index = 0; index < node.childNodes.length; ++index)
        result += getInnerText(node.childNodes.item(index));
    return result;
} // getInnerText
```

Attention, pour que cet exemple fonctionne sur MSIE, il faudra définir `Node` et ses constantes. Vous trouverez dans l'archive des codes source pour cet ouvrage (disponible sur le site des éditions Eyrolles) une démonstration complète de cette fonction.

DOMImplementation

Cette interface, accessible par la propriété `implementation` de l'objet global `document`, représente l'état de la prise en charge du DOM par le navigateur. Elle est utile pour vérifier qu'une fonctionnalité DOM est présente ou non.

De nombreux exemples et didacticiels sur Internet testent plutôt qu'une fonctionnalité est présente en vérifiant l'existence d'une fonction spécifique. Par exemple, pour vérifier qu'on dispose du DOM niveau 2, de nombreux scripts écrivent :

```
| if (document.getElementById && document.createTextNode)
```

Ce n'est pas mal, mais cela n'est pas toujours possible pour toutes les fonctionnalités susceptibles de nous intéresser. La façon générique de procéder consiste à utiliser la méthode `hasFeature` de cette interface, la seule qui nous intéresse. Cette méthode prend deux arguments : le nom officiel de la fonctionnalité et le niveau de DOM souhaité (une fonctionnalité pouvant évoluer d'un niveau à l'autre). L'argument de niveau peut valoir la chaîne vide ('') voire `null`, auquel cas il n'est pas pris en compte.

Les fonctionnalités sont définies de façon dispersée dans les spécifications du DOM et évoluent à chaque niveau. Elles correspondent principalement à des modules, et on trouve de-ci de-là dans la spécification une mention comme « Une application DOM peut utiliser la méthode `DOMImplementation.hasFeature(feature, version)` avec comme arguments "XML" et "3.0" (respectivement) pour déterminer si ce module est supporté ou non par l'implémentation. »

Ainsi, pour vérifier qu'on a le niveau 2, on peut écrire :

```
| if (document.implementation.hasFeature('Core', '2.0'))
```

D'accord, ce n'est pas plus court, mais primo, c'est valable pour tout test de fonctionnalité, et secundo, c'est tout de même plus lisible qu'une série de tests d'existence de méthodes.

Vous l'aurez compris, le paramètre `version` peut valoir `null`, '', '1.0', '2.0' ou '3.0'. Les valeurs utiles pour le paramètre `feature` incluent.

Tableau 3-10 Identifiants de fonctionnalités pour `hasFeature`

Fonctionnalité	Description
AS-READ, AS-EDIT, LS-AS	Sous-modules de Abstract Schemas (version 3.0).
Core	Fonctionnalités noyau (versions 2.0 ou 3.0, la 1.0 est garantie...).
CSS	Sous-module de style (version 2.0).
Events	Gestion événementielle (versions 2.0 ou 3.0).
HTML	Module HTML (versions 1.0 et 2.0).

Tableau 3-10 Identifiants de fonctionnalités pour hasFeature (suite)

Fonctionnalité	Description
LS, LS-Async	Sous-modules de Load and Save (version 3.0).
Range	Sous-module de Traversal and Range (version 3.0).
StyleSheets	Sous-module de style (version 2.0).
Traversal	Sous-module de Traversal and Range (version 3.0).
Validation	Validation de conformité aux grammaires (version 3.0).
Views	Manipulation des représentations visuelles (version 2.0).
ViewsAndFormatting, VisualViewsAndFormatting	Sous-modules d'extension de Views (version 3.0).
XML	Extensions XML (version 3.0).
XPath	Prise en charge de XPath (et souvent de XSLT ; version 3.0).

HTMLDocument

Nous entrons maintenant dans l'univers merveilleux du DOM HTML. Pourquoi merveilleux ? Parce qu'il va considérablement nous simplifier l'accès aux attributs HTML, pour commencer. Comme je l'ai déjà signalé, nous n'utiliserons pratiquement jamais `attributes`, ni même `getAttribute` ou `setAttribute` ! Le DOM HTML définit une interface spécifique pour chaque balise HTML officielle, avec des propriétés pour chaque attribut, et aussi des méthodes dédiées. C'est une mine d'or pour vos scripts, aussi ajoutez dès à présent la spécification à vos marque-pages : <http://www.w3.org/TR/DOM-Level-2-HTML/html.html>

Vous noterez que le DOM niveau 3 n'a pas touché au module HTML. On a donc ici une spécification stable, qui a subi pas moins de cinq révisions mais est solidifiée depuis près de quatre ans. Il n'est pas étonnant qu'elle soit bien prise en charge par la majorité des navigateurs.

Mais nous verrons quelques exemples de cela plutôt avec l'interface suivante, `HTMLElement`. Pour l'instant, examinons `HTMLDocument`, la spécialisation de `Document` dans le contexte des pages web.

Au risque d'en choquer certains, j'affirme pour commencer que `HTMLDocument` n'ajoute rien de véritablement utile à `Document`. Sur les 11 propriétés et 5 méthodes, je ne trouve une réelle utilité qu'aux propriétés `title` et `body`. C'est tout. Permettez-moi tout de même de justifier cette opinion :

- `referrer` est bien plus utile côté serveur ; côté client, elle n'ouvre la voie qu'à des manipulations tordues de l'historique ou à du *tracking* de navigation de l'utilisateur, deux utilisations un peu douteuses...

- `domain` et `URL` me semblent inutiles la majorité du temps : avez-vous souvent besoin qu'on vous dise sur quelle page vous êtes ? Et `domain` est trop souvent utilisé pour contourner le modèle de sécurité de JavaScript.
- Les collections `images`, `applets`, `links`, `forms` et `anchors` reviennent à de simples appels à `getElementsByName` (ou peu s'en faut).
- `cookie`, comme `referrer`, devrait plutôt être traité par la couche serveur, sans parler de la complexité inutile de sa manipulation en texte.
- `open()`, `close()`, `write(...)` et `writeln(...)` appartiennent au Crétacé du Web et ne devraient jamais être utilisées maintenant qu'on a le DOM et, souvent, `innerHTML`.
- `getElementsByName` a perdu 99 % de son intérêt avec XHTML ; elle est avantageusement remplacée par un ou plusieurs appels ciblés à `getElementById`.

Voilà ! Du coup, il ne nous reste que deux propriétés qui me semblent utiles dans des scripts modernes.

Tableau 3-11 Les deux propriétés vraiment utiles de `HTMLDocument`

Propriété	Description
<code>body</code>	Référence directe sur l'élément <code><body></code> du document. Il est fréquent d'avoir besoin de cet élément (pour parcourir son contenu ou ajouter un fragment en début ou fin de document), ce qui nous économise un <code>getElementsByName('body').item(0)</code> , ce qui n'est pas rien...
<code>title</code>	Le titre de la page, en lecture/écriture. De nombreux scripts ajustent le titre pour refléter un état, ce qui permet de le signifier partout où le système d'exploitation reprend ce titre : barre de titre de la fenêtre, titre du bouton dans la barre des tâches, etc.

Si vous examinez l'interface `HTMLDocument` dans la spécification, vous remarquerez que les propriétés collections utilisent comme type `HTMLCollection` plutôt que `NodeList`. L'utilisation est strictement compatible (propriété `length` et méthode `item`, si si), mais `HTMLCollection` a en plus une méthode `namedItem`, qui prend un ID d'élément et renvoie l'élément dans la collection dont l'attribut `id` a cette valeur, ou `null` en cas d'échec. Une sorte de `getElementById` localisé.

HTMLElement

Enfin, voici l'interface qui spécialise `Element` pour les éléments de pages web. Elle est elle-même spécialisée par une interface pour chaque balise HTML officielle : on trouve par exemple des interfaces `HTMLFormElement`, `HTMLInputElement`, `HTMLHeadingElement`, etc. Prenez le temps d'aller les découvrir dans la spécification et d'utiliser leurs nombreuses propriétés dédiées, qui simplifient grandement le code.

L'interface `HTMLElement` définit simplement cinq propriétés communes, qui correspondent à ce que la DTD de HTML appelle les « attributs noyau ». Voici ces cinq propriétés, dont on se sert tout le temps :

Tableau 3–12 Propriétés communes à tous les éléments HTML

Propriété	Description
<code>className</code>	Équivalent de l'attribut HTML <code>class</code> . Ne peut s'appeler <code>class</code> , parce que ce dernier est un mot réservé en JavaScript. Contient donc un ou plusieurs noms de classes, séparés par des espaces.
<code>dir</code>	Équivalent de l'attribut HTML <code>dir</code> . Peu utilisée dans des pages à langue unique, ou en tout cas ne mélangeant pas langues occidentales et orientales : indique la direction (de gauche à droite ou de droite à gauche) du texte dans l'élément.
<code>id</code>	Ai-je vraiment besoin de vous expliquer celle-ci ? Je rappelle qu'un ID est unique dans tout le document.
<code>lang</code>	Équivalent de l'attribut HTML <code>lang</code> . Spécifie donc la langue du texte contenu, <i>via</i> un code RFC 1766 (comme. <code>fr-FR</code>).
<code>title</code>	Équivalent de l'attribut HTML <code>title</code> . Fournit donc le texte alternatif de l'élément, qui apparaît généralement dans une infobulle lorsqu'on laisse le curseur de la souris un bref instant sur l'élément.

Eh bien voilà ! Nous avons vu l'essentiel des interfaces du DOM et du DOM HTML. Le reste est à examiner dans la spécification (encore une fois, si le format de cette dernière vous égare, jetez donc d'abord un œil à l'annexe C).

Nous réaliserons quelques exemples concrets et parfois copieux plus loin dans ce chapitre, s'articulant autour de besoins fréquents.

Quelques bonnes habitudes

Lorsqu'on travaille avec le DOM, c'est comme pour tout : il y a quelques bonnes façons de faire et beaucoup de mauvaises. Les conseils de cette section devraient déjà vous éviter une bonne partie des écueils et problèmes de maintenance.

Détecter le niveau de DOM disponible

Comme on l'a vu en parlant de l'interface `DOMImplementation`, de nombreux scripts détectent qu'ils disposent du DOM niveau 2 avec un code du style :

```
| if (document.getElementById && document.createTextNode)
```

En effet, vous remarquez qu'on n'a pas ajouté les parenthèses d'appel derrière les noms des méthodes : on récupère donc juste les objets `Function` correspondants. Si ces méthodes existent, ces objets seront différents de `null`, et donc, traités comme des booléens, équivalents à `true`. La condition sera alors validée.

Vous vous demandez peut-être pourquoi on teste deux méthodes, alors que `getElementById` suffirait (elle a été introduite au niveau 2). C'est parce que certains navigateurs, dans d'anciennes versions, ne fournissaient qu'une prise en charge très partielle du DOM niveau 2, qui proposait généralement `getElementById` en raison de son immense popularité, mais plus rarement des fonctions comme `createTextNode`. Il s'agit donc d'une tentative vague de protection contre une prise en charge trop partielle.

C'est généralement fiable, mais ce n'est pas forcément extensible à tous les besoins. Il peut arriver qu'un module DOM, à un niveau précis, n'introduise aucune nouvelle propriété ou méthode (il peut se contenter de modifier le comportement de méthodes ou propriétés, en ajoutant des cas d'erreur, en limitant leurs résultats, ou en activant la sensibilité à la casse par exemple). Que faire alors ?

On l'a vu, la solution réside dans l'emploi de la méthode `hasFeature` de `DOMImplementation`. Comment récupérer un objet proposant cette interface ? C'est très simple : une implémentation conforme de `Document` doit fournir une propriété `implementation`, qui sert précisément à cela. Quant à l'interface `Document`, on a vu qu'elle était proposée par l'objet global JavaScript `document`. Et cela fonctionne en effet sur tous les principaux navigateurs.

Pour détecter un niveau général de DOM, on utilisera la fonctionnalité `Core`. On écrira donc pour détecter le support de DOM niveau 2 ou ultérieur :

```
| if (document.implementation.hasFeature('Core', '2.0'))
```

Pour détecter la prise en charge de DOM niveau 3 XPath :

```
| if (document.implementation.hasFeature('XPath', '3.0'))
```

C'est simple, et surtout lisible et explicite !

Créer les nœuds dans le bon ordre

Lorsqu'on crée un fragment DOM, c'est-à-dire une série de nœuds qu'on va imbriquer les uns dans les autres (ce qui peut être aussi bête qu'un élément avec du texte à l'intérieur !), on doit garder à l'esprit deux considérations :

- Intuitivement, les éléments doivent être créés de l'extérieur vers l'intérieur : on va créer l'élément avant de créer le nœud texte à placer dedans, par exemple. On va créer la liste avant de créer ses éléments, etc.
- En revanche, chaque fois qu'un nœud est ajouté dans un autre, et que ce nœud conteneur est attaché au DOM, le navigateur est susceptible de déclencher immédiatement un *reflow*, c'est-à-dire une mise à jour de l'affichage. Cette mise à jour entraînera peut-être des décalages tous azimuts. Par exemple, lorsqu'une ligne devient plus longue que la largeur disponible, le paragraphe auquel elle appartient occupe tout à coup une ligne supplémentaire, ce qui décale le reste...

Il est bon de suivre le premier point et de créer les éléments de l'extérieur vers l'intérieur. En revanche, pour composer les éléments ensemble, généralement à coup d'appels à `appendChild`, plus rarement à `insertBefore`, il est préférable d'attendre que le fragment soit complet pour ajouter l'élément racine du fragment (l'élément externe, si vous préférez) au DOM de la page.

De cette façon, on ne causera qu'un seul reflow, évitant ainsi d'éventuels effets de bord disgracieux au fil de la construction du fragment. Qui plus est, plusieurs reflows auraient un impact négatif sur la vitesse d'exécution du script. En les évitant, on gagne donc tant sur le plan esthétique qu'en rapidité !

Ne scripter qu'après que le DOM voulu est construit

On touche ici à une source d'erreur fréquente chez les débutants qui scriptent le DOM. Il existe une règle simple et évidente : tant que le navigateur n'a pas lu et interprété un fragment donné de votre HTML, celui-ci n'est pas présent dans le DOM de la page.

Corollaire nécessaire : un script situé avant un élément dans le source HTML ne pourra pas immédiatement accéder à cet élément *via* le DOM. Dans la mesure où la salutaire séparation du contenu et du comportement impose d'utiliser uniquement des scripts externes, liés à la page à l'aide d'une balise du type :

```
<script type="text/javascript" src="chemin/fichier.js"></script>
```

et que ces balises sont par convention placées dans l'élément `<head>`, situé avant l'élément `<body>`, on a donc virtuellement la garantie que nos scripts seront analysés avant qu'il existe le moindre DOM pour le corps du document.

Par conséquent, un script qui contiendrait au niveau racine (c'est-à-dire hors de toute fonction) un code du style :

```
var header = document.getElementById('main-title');
```

est voué à l'échec : le HTML correspondant, probablement quelque chose du style `<h1 id="main-title">...</h1>`, n'a pas encore été traité à ce moment-là. Le navigateur traite en effet votre script au moment où il le charge, c'est-à-dire en traitant l'élément `<script>` qui l'invoque.

La solution est simple : faire une fonction pour votre code d'initialisation, et demander au navigateur d'appeler cette fonction une fois le DOM chargé. Par définition, à ce moment-là, l'ensemble du document aura été chargé, et sera donc représenté dans le DOM.

Idéalement, il faut réagir dès que le DOM est chargé, ce qui survient bien avant que la page elle-même ne le soit : entre les deux, le navigateur doit charger toutes les ressources de la page (CSS, images, applets, etc.). Ce chargement complémentaire est potentiellement très long et cela peut retarder d'autant l'exécution de vos scripts d'initialisation. Hélas, certains navigateurs ne fournissent un événement que pour ce dernier chargement, le plus tardif.

Ce chargement « portable » correspond à l'événement `onload` de l'objet global `window`. Suivant ce que vous avez sous la main pour écrire votre script, y attacher votre fonction sera plus ou moins compliqué. Nous verrons tout à l'heure les détails de la gestion événementielle, mais sachez que cela n'a rien de foncièrement difficile. Si vous disposez de la bibliothèque Prototype par exemple (que nous verrons en détail au chapitre suivant), cela revient simplement à écrire :

```
| Event.observe(window, 'load', myInitFunc, false);
```

Si vous n'avez pas Prototype mais avez la garantie d'un navigateur conforme au standard DOM niveau 2 événements (ce qui n'est pas le cas de MSIE, déjà...), c'est du même ordre :

```
| window.addEventListener('load', myInitFunc, false);
```

Encore une fois, un peu de patience : nous verrons les détails un peu plus tard dans ce chapitre.

Pour ceux qui ont un besoin impérieux d'exécuter du script dès le DOM chargé, sans attendre le chargement de la page, la bibliothèque Prototype garantit un événement synthétique nommé `dom:loaded`, que nous verrons au prochain chapitre.

Ne jamais utiliser d'extension propriétaire

Le Web est rempli d'articles, didacticiels et démonstrations écrits par des personnes qui n'ont pas forcément eu à cœur de s'en tenir aux standards. Souvent, « ça a marché pour eux, donc ça doit marcher pour vous ». Le problème, c'est que suite à la guerre des navigateurs des années 1990, chaque navigateur a mis en place des dizaines d'extensions propriétaires un peu partout dans JavaScript et le DOM, alors au fameux niveau zéro.

Lorsque vous tombez sur un script qui semble résoudre votre problème, commencez par vérifier scrupuleusement que toute portion technique dont vous ne maîtrisez pas le contenu (nom de propriété ou de méthode inconnue, etc.) est en réalité conforme aux spécifications. Il vous suffit de jeter un œil au standard de JavaScript et aux spécifications du DOM pour être fixé(e).

Bien sûr, on n'a parfois pas le choix : ainsi, lorsqu'une fonction standard n'est pas prise en charge par un navigateur, ni simulable en combinant d'autres fonctions standards, il faut bien recourir à la fonction propriétaire équivalente. Mais ces cas sont assez rares et correspondent presque toujours à une implémentation partielle ou incorrecte du DOM, généralement par MSIE.

Évitez par exemple d'utiliser `document.layers`, la fonction `GetObject`, la classe `ActiveXObject`, les commentaires conditionnels, etc. Souvenez-vous : le Web sera standard, ou ne sera plus !

Utiliser un inspecteur DOM

Il est temps de parler des outils. Nous l'avons déjà vu au chapitre 2, un bon outil d'aide au développement peut vous faire économiser un temps précieux. Lorsqu'il s'agit par exemple de déboguer du JavaScript, on peut choisir de passer la journée à traquer un problème subtil à coup de `alert`, ou de recourir à un débogueur intégré et de faire du pas à pas, pour trouver le souci en quelques minutes.

Un inspecteur DOM sert à afficher tout ou partie du DOM d'un document, sous forme d'une arborescence dépliable de nœuds. Il est bien entendu très utile pour mettre au point un script censé parcourir le DOM d'un document, puisqu'il évite de travailler à l'aveugle ou de recourir à une pléthore d'appels à `alert`.

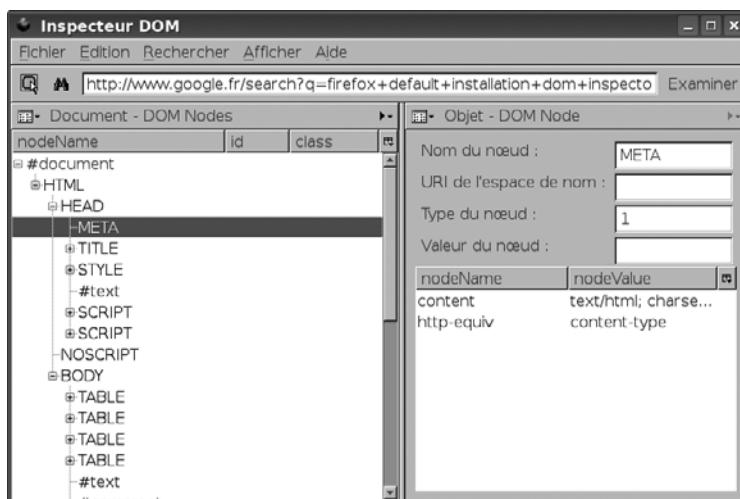
L'inspecteur DOM de Firefox/Mozilla

Nous avons vu au précédent chapitre le débogueur « officiel » de Mozilla, baptisé Venkman. Il est mis à disposition sous forme d'une extension. L'inspecteur DOM, en revanche, fait partie intégrante du navigateur. Pour l'obtenir, il vous faut procéder différemment suivant votre plate-forme.

- Sous Windows (et, je crois, Mac OS X), il faut l'avoir installé explicitement, ce qui est à mon sens une bêtise de la part de la fondation Mozilla. Si vous n'aviez pas opté pour une installation personnalisée puis coché l'option, vous en êtes quitte pour relancer l'installation après avoir fermé toutes vos fenêtres Firefox. Il vous faudra alors choisir le mode personnalisé, et cocher lorsqu'on vous le proposera l'option **Inspecteur DOM**. Soyez tranquille, vous n'aurez pas de doublon dans la liste des programmes installés sur le disque, et vous conserverez tout votre profil d'utilisateur.
- Sous Linux, suivant votre distribution, il fera ou non partie du paquet **firefox** (**iceweasel** sur Debian). À vous de voir si vous disposez, dans le menu Outils, d'une option **Inspecteur DOM**. Dans la négative, recherchez le paquet idoine et installez-le après avoir fermé toutes vos fenêtres. Sur Debian par exemple, il s'appelle **iceweasel-dom-inspector**.

Une fois disponible, l'inspecteur s'obtient depuis le menu **Outils>Inspecteur DOM**, ou en pressant **Ctrl+Maj+I**. Il inspecte par défaut la page en cours, mais vous pouvez modifier ce comportement. Voici son aspect général.

Figure 3–2
L'inspecteur DOM de Mozilla



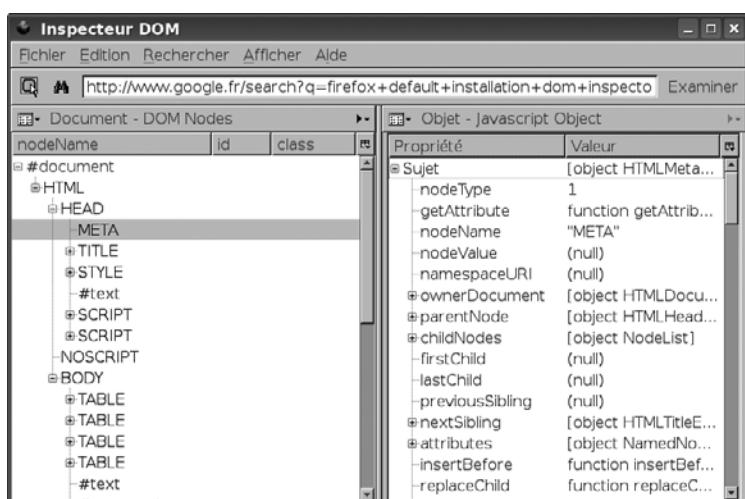
Sur la gauche, vous avez le DOM de la page, dépliable et navigable de façon classique. Sur la droite, vous avez, au choix, l'objet DOM pur (choix par défaut) ou l'objet DOM JavaScript correspondant au nœud sélectionné à gauche.

La figure 3–2 montre le DOM d'une page de résultats Google, avec un nœud sélectionné qui propose plusieurs informations dans la vue du DOM pur : le nom du nœud (ici sa balise canonique, puisqu'il s'agit d'un élément dans un document en mode HTML), son type (1, donc `Node.ELEMENT_NODE`), sa valeur (vide comme pour tout élément) et ses attributs.

On a parfois moins de chance, par exemple sur des nœuds texte, où tout ceci est remplacé par une vaste zone de texte, ce qui pour les nombreux nœuds texte vides n'est pas d'une grande utilité.

Prenons le panneau de droite. En haut à gauche de ce panneau, une icône permet, en cliquant dessus, d'afficher une liste déroulante, dans laquelle vous pouvez choisir l'option JavaScript Object. La liste qui s'affiche alors offre une vue bien plus détaillée du nœud courant ; c'est pratique pour aller dénicher l'information, mais cela peut aussi nous submerger. Par exemple, pour le même nœud que dans la figure précédente, on obtient ici, après avoir déroulé le premier élément de la liste, nommé « Sujet ».

Figure 3–3
La vue JavaScript Object
pour le même nœud



C'est sans surprise : on a l'ensemble des propriétés et méthodes définies par la spécification du DOM. Vous avez peut-être remarqué qu'elles ne semblent pas être triées dans l'ordre alphabétique. Elles sont plus ou moins exactement groupées par niveau, module, interface et enfin ordre de définition dans la spécification. Ici, on a d'abord toutes celles de `Node` (`nodeType`, `nodeName`...), puis celles de `Element` (`attributes`, `getAttribute`, qui fait d'ailleurs exception ici en étant présent plus tôt), et plus bas, hors de l'image capturée ici, celles de `HTMLElement` (`id`, `title`...) et celles de `HTMLMetaElement` (`content`, `http_equiv`). On trouve ensuite, les propriétés issues du module `Style` et les ajouts effectués à `Node` par le DOM niveau 3.

Voici un rapide tour des menus et services fournis :

- Le menu Fichier vous permet d'inspecter un autre document, soit en sélectionnant une des fenêtres ouvertes de votre navigateur, soit en précisant une nouvelle URL.
- Le menu Rechercher permet de chercher dans le DOM par nom de nœud, valeur d'ID ou valeur pour un attribut précis.

- Le menu Afficher permet de restreindre les types de nœuds affichés (en omettant les nœuds vides, par exemple), mais aussi de choisir si la vue navigateur met en exergue l'élément visuel sélectionné dans l'arborescence DOM (comportement par défaut) ou non. En effet, par défaut, sélectionner un élément dans l'arbre du DOM qui est affiché dans la page produit un cadre clignotant l'espace d'une seconde autour de cet élément, pour aider le développeur à s'y repérer.

L'inspecteur DOM de Firebug

Au chapitre 5, nous verrons les capacités de débogage JavaScript de l'extension Firebug pour Firefox. Mais Firebug ne se limite pas, loin s'en faut, à un débogueur et une console. Firebug propose aussi un inspecteur, et même plusieurs, accessibles en choisissant le bouton *Inspect* ou l'onglet *Inspector* :

- Un inspecteur *Source*, qui affiche un arbre des éléments avec leur représentation HTML (et une coloration syntaxique, s'il vous plaît !). Survoler un élément du document le sélectionne automatiquement.
- Un inspecteur *Style*, qui permet d'afficher les styles, explicites (attributs `style`) ou complets (c'est-à-dire incluant les valeurs par défaut et celles provenant de règles CSS), pour l'élément survolé. La bascule se trouve dans le menu Options de la barre Firebug.
- Un inspecteur *Layout*, qui affiche l'ensemble des propriétés relatives au positionnement pour l'élément survolé. Précieux quand on travaille sur du code à la script.aculo.us !
- Un inspecteur *Events*, qui affiche les événements lorsqu'ils se déclenchent.
- Enfin, un inspecteur *DOM*, qui peut fonctionner en mode global (affiche tous les objets globaux de la page et les propriétés de l'objet courant, donc `window`) ou en mode survol (affiche le DOM de l'objet survolé). On bascule de l'un à l'autre avec le bouton *Inspect* de la barre Firebug.

Les figures suivantes présentent quelques exemples d'aspect pour ces inspecteurs.

Vous trouverez de nombreux exemples supplémentaires (et impressionnantes !) sur la page dédiée du site officiel : <http://joehewitt.com/software/firebug/screens.php>. Cliquez sur les vignettes pour voir les captures d'écran complètes.

Personnellement, j'utilise bien plus souvent l'inspecteur DOM de Firebug que celui de Firefox. Je vous conseille d'utiliser les deux quelque temps pour déterminer celui qui vous semble le plus agréable.

Figure 3–4

L'inspecteur Source avec le nœud title sélectionné

**Figure 3–5**

L'inspecteur Style au survol d'une image que son attribut style rend flottante

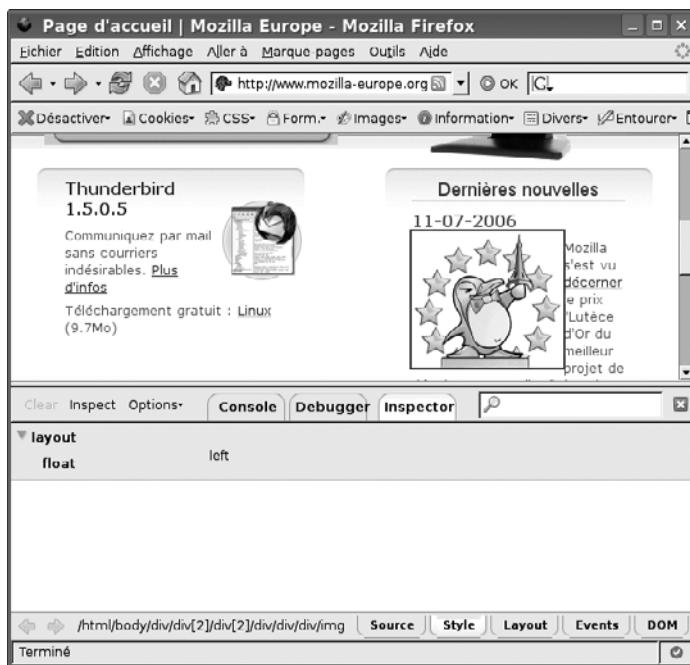
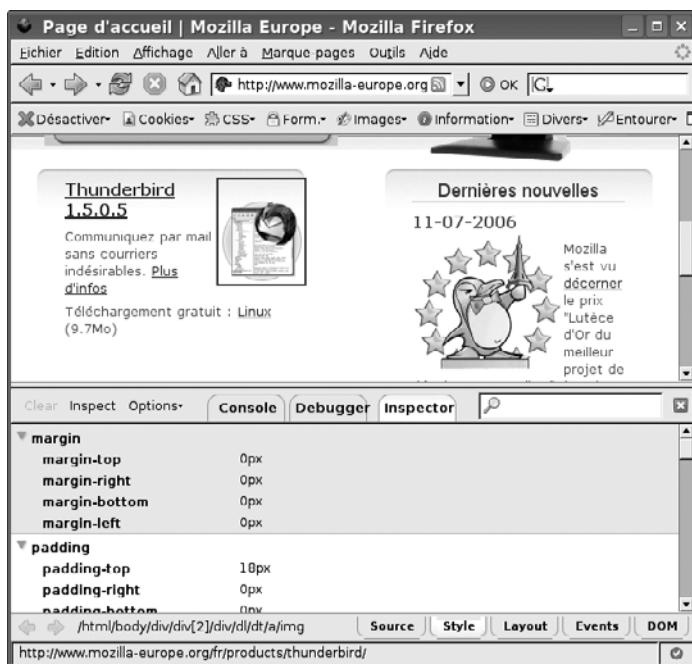


Figure 3–6

L'inspecteur Style en mode « styles complets » sur une autre image

**Figure 3–7**

L'inspecteur Layout sur un titre : deux colonnes de styles et une de propriétés supplémentaires relatives au positionnement

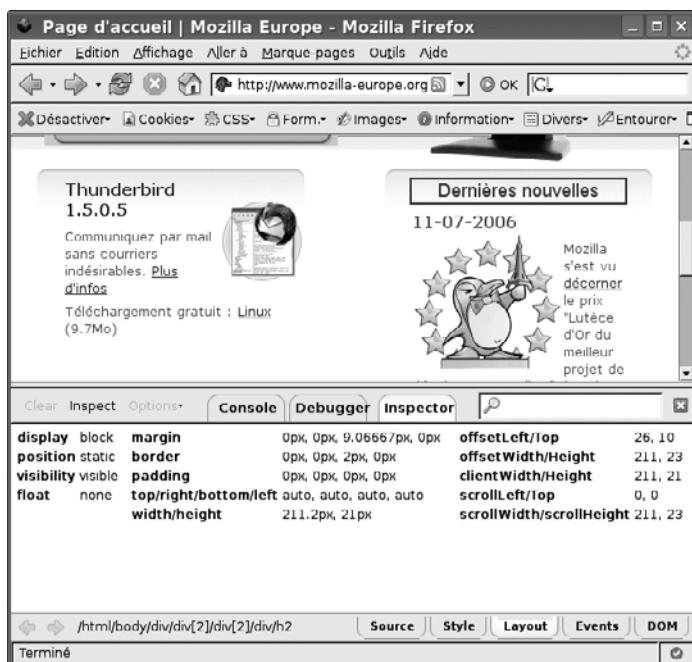


Figure 3–8
L'inspecteur DOM
en mode global



Figure 3–9
L'inspecteur DOM en mode survol, sur un lien



Répondre aux événements

Après ces quelques conseils, abordons le dernier point technique du DOM que nous n'avons pas vu : les événements. Sans gestion événementielle, c'est bien simple : votre page est morte. C'est-à-dire, pour faire une lapalissade, qu'elle n'est pas vivante. Elle ne réagit pas à la souris, au clavier, ni même aux événements internes ou basés sur des *timers*. La page est le bec dans l'eau.

Nous avons déjà évoqué brièvement la gestion événementielle. Ce sujet se découpe en deux grandes tâches :

- 1 Associer un gestionnaire (une fonction que nous avons écrite) à un événement précis pour un objet précis.
- 2 Traiter cet événement lorsqu'il survient.

En raison de la fameuse guerre des navigateurs des années 1990, on a vu fleurir plusieurs manières totalement incompatibles entre elles de gérer des événements. Google n'oubliant jamais, on trouve encore de très nombreuses pages prônant l'une ou l'autre de ces méthodes aujourd'hui obsolètes, que ce soit pour associer l'événement à un objet ou pour traiter l'objet événement lorsqu'il survient.

Afin de vous aider à bien distinguer les principales approches et à comprendre en quoi toutes (sauf une !) posent problème, nous allons les étudier tour à tour. Il s'agit là de culture technique, car lorsque nous passerons à Prototype, au chapitre suivant, ces incompatibilités et ces détails techniques seront masqués.

Les truands : les attributs d'événement dans HTML

Mais si, vous les avez déjà vus : ce sont les attributs `onxxx` dans le HTML. Voici quelques exemples courants :

```
<body onload="initPage()">
...
<form method="post" action="process.php" onsubmit="checkForm()">
...
<a href="#" onclick="return popupWindow('help.html')">Aide</a>
...
<div onclick="doSomething()">
...
```

Certains de ces exemples cumulent les tares en enfreignant plusieurs règles techniques et éthiques... Tous posent en tout cas trois problèmes :

- 1 Ils représentent une intrusion du comportement dans le contenu.

2 Ils ne peuvent pas facilement associer plusieurs gestionnaires à un même événement pour un même élément.

3 Ils n'existent pas pour tous les événements de tous les éléments.

En somme, c'est à bannir, ne serait-ce que pour la première raison. La séparation stricte du contenu, de l'aspect et du comportement est un objectif permanent, une façon de travailler, un état d'esprit.

La brute : les propriétés d'événement dans le DOM niveau 0

« Très bien, » pensez-vous, « je vais déporter ces affectations dans mon JavaScript, au sein d'une fonction d'initialisation appelée après le chargement de la page, comme suggéré plus haut dans ce chapitre ».

L'idée est noble, et je vous en félicite avec un grand sourire, mais on peut tout de même mal l'exécuter. Par exemple, vous pourriez tomber dans le travers fréquent que voici :

Listing 3-5 Une association très imparfaite de gestionnaires d'événements

```
function initEventHandlers() {  
    document.getElementById('mainForm').onsubmit = checkForm;  
    document.getElementById('helpPopupLink').onclick = popupHelp;  
    ...  
} // initEventHandlers  
  
window.onload = initEventHandlers;
```

D'accord, vos gestionnaires d'événements sont désormais associés aux éléments dans le script, ce qui débarrasse votre HTML de tout attribut onxxx, n'y laissant plus que du contenu. Il y a effectivement du progrès.

Mais imaginez qu'après votre balise <script> chargeant ce fichier JavaScript, vous chargez un script supplémentaire qui, lui aussi, a besoin de lancer une de ses fonctions au chargement. S'il procède de la même façon :

```
window.onload = myInitFunction;
```

Votre propre initialisation passera aux oubliettes !

On le voit, cette façon de procéder partage un inconvénient avec celle vue précédemment : il n'est pas possible d'associer plusieurs gestionnaires au même événement pour un même élément. Les bogues qui en résultent peuvent être très difficiles à diagnostiquer correctement.

Par ailleurs, là aussi, tous les événements potentiels ne disposent pas toujours d'une propriété idoine.

Le bon : addEventListener

Voici enfin la bonne façon de faire, qui est d'ailleurs celle spécifiée par le standard DOM Level 2 Events (www.w3.org/TR/DOM-Level-2-Events/events.html).

Par définition, tout objet proposant l'interface `Node` constitue une cible d'événements potentielle et propose donc également l'interface `EventTarget`. Celle-ci fournit trois méthodes, dont deux nous intéressent particulièrement.

Tableau 3-13 Les deux méthodes clefs de l'interface `EventTarget`

Méthode	Description
<code>addEventListener(type, listener, useCapture)</code>	Ajoute (inscrit, si vous préférez) un gestionnaire d'événement.
<code>removeEventListener(type, listener, useCapture)</code>	Retire (désinscrit) un gestionnaire d'événement.

Les deux méthodes ont les mêmes arguments, ne renvoient rien et ne lèvent aucune exception. Là où les deux premiers arguments sont simples à comprendre, le troisième est plus délicat.

- `type` décrit l'événement concerné : il s'agit d'un nom, généralement en minuscules, sans le préfixe `on`. La spécification précise la liste des événements valides du niveau 2, ainsi que les anciens événements du niveau 0, pour lesquels une compatibilité est maintenue jusqu'à présent. Un tableau est fourni plus bas.
- `listener` référence simplement la fonction de traitement.
- `useCapture` indique qu'on souhaite capturer l'événement plutôt que l'intercepter lors de son bouillonnement. Ces notions un peu avancées sont présentées un peu plus loin dans ce chapitre. La plupart du temps, vous mettrez `false`.

Dans la majorité des cas, on ne prend pas la peine de désinscrire son gestionnaire, car celui-ci est valide pendant toute la durée de vie de la page. On utilise donc principalement `addEventListener`. Voici la version propre du listing précédent.

Listing 3-6 Une association de gestionnaires conforme aux standards

```
function initEventHandlers() {
    document.getElementById('mainForm').addEventListener(
        'submit', checkForm, false);
    document.getElementById('helpPopupLink').addEventListener(
        'click', popupHelp, false);
    ...
} // initEventHandlers

window.addEventListener('load', initEventHandlers, false);
```

Vous remarquerez que ce n'est guère plus compliqué, ni moins lisible. Avant d'entrer dans les détails sordides de la compatibilité avec MSIE et de cette histoire de capture et de bouillonnement, dressons une liste des principaux événements reconnus, décrits à la section 1.6 de la spécification :

Tableau 3-14 Principaux événements reconnus par le DOM niveau 2

Catégorie	Type	Description
UI (interface utilisateur)	DOMFocusIn	Équivalent de l'ancien onFocus : l'élément a reçu le focus clavier.
	DOMFocusOut	Équivalent de l'ancien onBlur : l'élément a perdu le focus clavier.
	DOMActivate	L'élément a été activé. La propriété detail de l'objet Event indique alors s'il s'agit d'une activation simple (1 : clic, touche Entrée) ou double (2 : double-clic, Maj+Entrée).
Souris	click	Clic de souris (ou équivalent clavier) : enfoncement puis relâchement d'un bouton.
	mousedown	Enfoncement d'un bouton.
	mouseup	Relâchement d'un bouton.
	mouseover	La souris commence à survoler l'élément (« entrée dans l'espace aérien »).
	mousemove	La souris survole l'élément.
	mouseout	La souris vient de cesser de survoler l'élément (« sortie de l'espace aérien »).
Clavier	Pas encore spécifiés, c'est ahurissant ! On se rabat pour l'instant sur les événements niveau 0, à savoir keypress, keydown et keyup.	

Chaque type d'événement utilise une version spécialisée de Event pour passer les détails au gestionnaire. Examinez la spécification pour obtenir la liste des propriétés spécifiques à chaque type.

Les événements compatibles DOM niveau 0 sont pour le moment toujours autorisés ; la spécification les nomme « événements HTML » ! En voici la liste.

Tableau 3-15 Événements de compatibilité avec le DOM niveau 0

Type	Description
load	Chargement terminé du document, de l'objet ou du cadre. Sans équivalent de niveau 2.
unload	La fenêtre ou le cadre va se fermer. À utiliser avec circonspection, peut s'avérer irritant ! Sans équivalent non plus (et c'est tant mieux !).
abort	Interruption volontaire du chargement du document, de l'objet ou du cadre.
error	Une image n'a pu se charger ou une erreur est survenue dans un script.
select	Du texte a été sélectionné dans le champ de saisie.
change	La valeur du champ a changé (déclenché à sa perte de focus uniquement, sauf, souvent, pour les select).
submit	L'utilisateur demande à envoyer le formulaire.

Tableau 3-15 Événements de compatibilité avec le DOM niveau 0 (suite)

Type	Description
reset	Le formulaire est réinitialisé (retour aux valeurs spécifiées dans le HTML).
focus	Le champ ou libellé (<code>label</code> , <code>input</code> , <code>select</code> , <code>textarea</code> , <code>button</code>) vient de récupérer le focus clavier.
blur	Le champ ou libellé a perdu le focus clavier.
resize	La fenêtre (ou en tout cas la vue contenant le document) est redimensionnée.
scroll	La fenêtre (...) subit un défilement.

Évitez toutefois de les utiliser lorsqu'un équivalent de niveau 2 existe, ne serait-ce que pour la pérennité de votre code...

Accommoder MSIE

Eh oui, on y revient toujours : alors que `addEventListener`, la méthode officielle, dont la spécification finale a déjà 6 ans, est prise en charge par l'ensemble des navigateurs répandus, MSIE n'en a jamais entendu parler. Non ! Dans MSIE, mes chers lecteurs, on utilise `attachEvent`. D'où cela sort-il ? De l'imagination féconde, quoique mal avisée, des développeurs du siècle dernier (littéralement). Remarquez, le nom a du sens, c'est déjà ça.

Cette méthode déviante est décrite sur <http://msdn.microsoft.com/workshop/author/dhtml/reference/methods/attachevent.asp>, et si vous allez y jeter un œil, vous verrez qu'en dépit d'un certain culot consistant à nommer la liste propriétaire Microsoft des noms d'événements une « liste des événements DHTML *standards* », l'interface est simple : le nom d'événement, et la fonction de gestion. Pas de troisième argument en revanche, MSIE n'offrant pas de mécanisme de capture, mais uniquement le bouillonnement. Et les noms d'événements *standards* (quel humour, ces rédacteurs de documentation !) sont précédés du traditionnel préfixe `on`, comme au bon vieux temps.

La traduction est donc simple : dans MSIE, au lieu de faire :

```
| node.addEventListener('event', handler, false);
```

On fait :

```
| node.attachEvent('onevent', handler);
```

Rien de bien sorcier, mais si on pouvait éviter de coder le `if/else` à chaque association, ce serait mieux. On va donc écrire une fonction pour cela.

L'idéal serait de la rendre disponible dans tous les objets, y compris tous les objets natifs du DOM, pour la manipuler de façon très similaire à `addEventListener`, mais hélas c'est impossible : la plupart des objets natifs ne fournissent pas de prototype sur lequel greffer notre méthode. Il faut donc se contenter d'une bête fonction, à l'ancienne. Voici un exemple d'implémentation.

Listing 3-7 Un exemple de fonction portable d'association de gestionnaire

```
function addListener(element, baseName, handler) {  
    if (element.addEventListener)  
        element.addEventListener(baseName, handler, false);  
    else if (element.attachEvent)  
        element.attachEvent('on' + baseName, handler);  
} // addListener
```

Notez que le cas restant (ni `addEventListener`, ni `attachEvent`) concerne si peu de navigateurs, tous un peu marginaux, qu'il ne mérite pas qu'on s'y intéresse, surtout qu'alors il n'est généralement pas possible de scripter la gestion événementielle !

J'ai mis cette fonction à titre d'exemple, et pour vous faire comprendre la mécanique. Dans la pratique, dès le prochain chapitre, on utilisera `Event.observe`, de Prototype, qui fait fondamentalement la même chose (avec tout de même plein de petits détails autour).

La propagation : capture ou bouillonnement ?

Depuis tout à l'heure, vous lisez partout capture et bouillonnement. Si vous n'êtes pas habitué(e) aux mécanismes de propagation d'événements dans les navigateurs, ces termes vous sont étrangers (au moins un des deux).

Il s'agit des deux modes historiques d'interception des événements. Ils s'appliquent uniquement aux événements de l'interface utilisateur et non aux événements internes ou système, comme le déclenchement d'un *timer*.

Il faut bien comprendre une chose : les événements sont traités par le navigateur indépendamment de l'existence de gestionnaires. Que vous ayez ou non défini des gestionnaires associés à un événement, lorsque celui-ci survient, le navigateur crée toujours un objet pour le représenter et notifie les éléments concernés que l'événement est survenu, en leur passant l'objet qui le représente.

Dans les explications qui vont suivre, on se basera sur le document HTML d'exemple de la figure 3-8.

Listing 3-8 Un document HTML simple pour nos explications

```
<html>
...
<body>
  ...
    <div id="navbar">
      ...
        <a id="homeLink" href="/home">Accueil</a>
      ...
    </div>
  </body>
</html>
```

Le modèle le plus courant : le bouillonnement

La plupart des événements bouillonnent. Pas tous ceci dit, vérifiez au cas par cas dans la spécification (c'est précisé pour chaque type d'événement). Par exemple, les événements de niveau 0 `load`, `unload`, `focus` et `blur` ne bouillonnent pas (ce qui est logique, si on y réfléchit après avoir lu cette section).

Un événement bouillonnant (ce qui ne signifie pas qu'il fera la une des journaux) est d'abord déclenché sur l'élément le plus proche d'après le contexte courant ; en clair, l'élément situé sous la souris (pour un événement souris), ou ayant le focus clavier (pour un événement clavier). Cet élément, qui constitue la cible physique de l'événement, est souvent appelé l'élément source. L'événement est ensuite déclenché à nouveau pour chaque nœud parent de l'élément source. L'événement remonte donc toute la hiérarchie du document, jusqu'au nœud racine `Document` lui-même. On dit que l'élément bouillonne, traduction un peu pataude pour le terme anglais *to bubble up*.

Ainsi, dans le document du listing 3-8, si l'utilisateur clique sur le lien `homeLink`, c'est d'abord ce lien qui va recevoir l'événement, puis son nœud parent, `navbar`, ensuite le `body`, l'élément `html`, et finalement `document`, l'objet global qui représente le nœud racine du DOM pour la page.

Ce qui est très important, c'est que chaque nœud sur ce chemin a l'opportunité de stopper la propagation, ou si vous préférez, d'interrompre le bouillonnement. La procédure standard pour cela consiste à appeler la méthode `stopPropagation()` de l'objet `Event` (et sous MSIE, c'est bien entendu différent : on met la propriété `cancelBubble` à `true`).

Pourquoi stopper la propagation ? C'est pratique lorsque vous savez que votre gestionnaire est censé être le « terminus » pour cet événement. Et c'est le cas la plupart du temps : les pages où un même événement doit être traité à plusieurs niveaux hiérarchiques sont rares.

La capture, ou comment jouer les censeurs

La capture n'est pas, contrairement à une idée répandue, juste l'inverse du bouillonnement. En réalité, les deux mécanismes peuvent parfaitement coexister.

La capture est conçue pour permettre à un gestionnaire placé à un niveau donné du DOM de « censurer » l'événement qu'il reçoit pour tout le fragment dont son noeud est racine (en tout cas, au moment de l'association : un noeud descendant à ce moment-là, mais qui serait ensuite déplacé hors du fragment par une manipulation du DOM, serait toujours sujet à la capture).

Il ne s'agit donc pas de traiter l'événement à proprement parler, mais de le censurer d'après un algorithme correspondant à votre logique d'interface, que vous aurez implémentée dans le gestionnaire. Lorsque l'événement visé a lieu pour un élément descendant de celui sur lequel vous avez enregistré la capture (je dis bien descendant : pas le noeud lui-même, ni un noeud ailleurs dans le DOM), votre gestionnaire est déclenché. S'il stoppe la propagation (toujours avec la méthode `stopPropagation` de l'objet `Event`), l'événement ne sera pas déclenché sur son élément source, et ne bouillonnera donc pas le cas échéant.

Ainsi, prenons l'appel suivant :

Listing 3-9 Un exemple de capture pour censure inconditionnelle

```
document.getElementById('navbar').addEventListener('click',
    function(event) { event.stopPropagation(); }, true);
```

Ce code empêcherait toute détection de clic par le contenu de `navbar`, et donc, entre autres, notre lien `homeLink`.

Il faut préciser que ce mécanisme est rarement utile. On s'en sert plus pour geler l'interface pendant un traitement (en refusant tout événement utilisateur jusqu'à nouvel ordre sur tout ou partie du document) que pour des besoins subtils et perfectionnés. Par ailleurs, certains événements ne sont pas capturables ; c'est le cas par exemple des événements relatifs au focus, de `mousemove` ou encore de `load` et `unload`.

Et pour finir, la cerise habituelle : le mécanisme de capture n'est pas pris en charge par MSIE. D'ailleurs, vous avez bien vu que `attachEvent` n'a pas de paramètre pour la capture, et vous cherchez en vain une méthode `captureEvent`, qui aurait trop ressemblé à son homonyme du Netscape de l'époque...

Pour résumer, faites simple : évitez la capture.

L'objet Event

Le déclenchement d'un événement donne lieu à la création d'un objet pour le représenter. Cet objet est censé implémenter l'interface `Event`, ainsi qu'une interface plus spécialisée selon le type de l'événement (par exemple, `UIEvent` ou `MouseEvent`). Je dis censé, car bien entendu, MSIE traite cela à sa façon.

Tout gestionnaire d'événement reçoit normalement l'objet `Event` en argument. Sous MSIE, il n'est pas passé en argument mais est présent dans l'objet global `window.event`.

Le module Événements est l'un des points les plus sensibles de la faille entre MSIE et le respect du DOM niveau 2. Cette faille est bien sûr masquée par des bibliothèques comme Prototype. Afin de simplifier la description et de l'homogénéiser, les sections qui suivent présenteront toujours trois syntaxes pour chaque aspect, toutes trois supposant que l'argument du gestionnaire s'appelle `event` :

- La syntaxe officielle du standard, nommée « DOM ».
- La syntaxe propriétaire de MSIE, seule prise en charge par ce dernier.
- La syntaxe portable offerte par Prototype, ce qui constitue certes un petit saut en avant vers le chapitre 4, mais vous rassurera à chaque fois quant à l'inutilité de devoir jongler manuellement entre les versions officielle et MSIE.

Récupérer l'élément déclencheur

Un gestionnaire a souvent besoin de récupérer l'élément source, ou cible, de l'événement déclenché. C'est particulièrement vrai quand vous déclarez un gestionnaire unique au niveau du conteneur pour traiter de façon similaire un même événement survenant chez plusieurs éléments descendants (si cet événement bouillonne, comme le fera par exemple un clic, votre gestionnaire en sera forcément notifié).

- DOM : `event.target`
- MSIE : `window.event.srcElement`
- Prototype : `Event.element(event)`

Stopper la propagation

Nous avons déjà examiné ce mécanisme à la section sur la capture et le bouillonnement.

- DOM : `event.stopPropagation()`
- MSIE : `window.event.cancelBubble = true`
- Prototype : `Event.stop(event)` (annule aussi le traitement par défaut)

Annuler le traitement par défaut

La notion de traitement par défaut est très intéressante et critique pour de nombreuses utilisations.

La majorité des événements ont un traitement par défaut, qui correspond à ce que ferait le navigateur si vous ne définissiez aucun gestionnaire pour l'événement. Par exemple, cliquer sur un lien navigue vers la cible de ce lien ; soumettre un formulaire envoie les informations à la couche serveur.

Sauf demande explicite dans votre gestionnaire, ce traitement par défaut aura lieu. Or, une fonction de vérification de validité des saisies dans un formulaire voudra pouvoir empêcher l'envoi du formulaire, en plus de signaler les problèmes de saisie. De même, une fonction chargée d'afficher la cible d'un lien dans une fenêtre surgissante voudra empêcher la fenêtre contenant le lien de naviguer elle aussi vers la cible.

Vous trouverez de nombreux exemples sur le Web qui vous diront qu'il suffit que votre gestionnaire renvoie `false`, ce qui est aujourd'hui parfaitement périmé : ça ne marche que pour des attributs d'événement incrustés dans le balisage, mais en aucun cas pour des gestionnaires d'événements à part (même sur MSIE).

Les mécanismes opérationnels à ce jour sont les suivants :

- DOM : `event.preventDefault()`
- MSIE : `window.event.returnValue = false`
- Prototype : `Event.stop(event)` (stoppe aussi la propagation).

JavaScript, événements et accessibilité

Une page qui ne fonctionnerait qu'avec JavaScript activé, parce qu'elle reposeraient intégralement sur JavaScript et les événements pour remplir son rôle, constituerait un grave problème d'accessibilité au sens large.

D'abord, de nombreux contextes n'auront pas JavaScript (navigateurs textuels, certains navigateurs sur périphériques mobiles : téléphones, Palm Pilot/Visor, etc.), auront JavaScript désactivé (par décision des responsables informatiques de l'entreprise), ou auront une prise en charge très partielle (lecteurs d'écran). Tous ces utilisateurs, pourtant parfaitement légitimes, ne pourront utiliser correctement la page.

Même avec JavaScript opérationnel, il est irresponsable d'exiger des manipulations clavier ou souris complexes de tous vos utilisateurs : certains souffrent peut-être d'un handicap moteur ou ont un périphérique (*touchpad* ou *trackpoint* sur un portable) les empêchant de manipuler la souris avec précision ; d'autres ne peuvent confortablement lui associer un modificateur clavier (par exemple, si vous attendez un *Ctrl+Clic*), voire tout simplement utiliser la souris (handicap moteur plus lourd). Utiliser des événements souris de façon exclusive est très vite limitatif.

Quand bien même vous décideriez d'envoyer paître la base utilisateur « handicapée » au sens large (handicaps moteurs, visuels, cognitifs), soit plus de 15 % des internautes mondiaux (plus d'un million de personnes rien qu'en France), vous n'aurez peut-être pas le choix : un niveau élevé d'accessibilité est aujourd'hui une exigence légale pour tout appel d'offres émanant du service public, et un nombre croissant d'appels d'offres privés l'exigent également.

Accessibilité ne rime pas avec impossibilité, ni même avec complexité : il s'agit simplement d'ajuster nos habitudes de développement pour l'intégrer dans nos réflexes de code. Vous trouverez une panoplie très complète de documentations pratiques en français traitant de l'accessibilité pour le Web sur le site d'Accessiweb, la cellule spécialisée de l'association BrailleNet : <http://www.accessiweb.org/>.

Voici déjà quelques conseils à retenir :

- Une pierre d'angle est bien sûr l'*unobtrusive JavaScript*, déjà discuté, qui consiste à ne jamais mettre de scripts ni d'attributs événementiels dans votre HTML : n'y laissez que le contenu !
- Autre réflexe important : pour assurer que votre page se dégrade élégamment (c'est-à-dire continue de fonctionner correctement au fur et à mesure que les moyens du bord se raréfient : plus de CSS, plus d'image, plus de JavaScript...), le mieux est de la réaliser par amélioration progressive.
En d'autres termes, commencez par faire une page capable de fonctionner sans aucun JavaScript, mais qui n'aura recours qu'à des allers-retours avec la couche serveur. Ensuite, améliorez-la par petites touches à coups d'ajouts de gestionnaires depuis votre fichier de script (toujours *unobtrusive...*). En partant du bas, vous garantissez que la page est capable d'y retourner !
- Les fonctions de confort réalisées en JavaScript n'ont pas obligatoirement à avoir un équivalent classique ; ce qui compte, c'est que la page puisse fonctionner, donc rendre le service qui est sa raison d'être, sans JavaScript. Tant pis si c'est alors, fatallement, un peu plus pénible, un peu plus ardu. L'exemple typique est le tri de liste : sans JavaScript, on doit faire un aller-retour à chaque déplacement d'un élément vers le haut ou vers le bas ; avec JavaScript, on peut glisser-déplacer tout ce beau monde vite fait et valider à la fin !

Malgré beaucoup d'astuce dans l'emploi de JavaScript, il reste toutefois difficile de rendre facile d'emploi, particulièrement au clavier, des interfaces dynamiques un peu riches, comme des arborescences dépliables et repliables, des menus déroulants à niveaux multiples ou encore des grilles de données.

Mais sur ce front, l'espoir renaît : pour faciliter l'accessibilité de ces réalisations, une initiative « DHTML accessible » a vu le jour conjointement entre le W3C, IBM et la fondation Mozilla. Elle est d'ores et déjà implémentée dans Firefox 1.5. À l'aide d'attributs supplémentaires décrivant le rôle fonctionnel des éléments, il est possible

de faciliter grandement la navigation et l'utilisation au clavier de composants visuels riches et complexes (principalement à l'aide des flèches et de Tabulation, comme sous Windows ou Mac OS X, par exemple). Cela ouvre des horizons ! Vous en trouverez davantage sur la page dédiée du site Mozilla Developer Connection (MDC) et sur les pages concernées du W3C :

- http://developer.mozilla.org/en/docs/Accessible_DHTML
- <http://www.w3.org/WAI/PF/roadmap/>

L'initiative se penche d'ailleurs aussi sur l'accessibilité d'Ajax. On ne peut que l'encourager !

Enfin, quelques techniques utiles :

- Les *accessible pop-ups*, ou fenêtre surgissantes accessibles, décrivent comment faire en sorte qu'un lien s'affiche dans une fenêtre surgissante si JavaScript est actif, ou suivre sa navigation traditionnelle dans le cas contraire. C'est un grand classique, décrit avec tous les détails de mise au point dans <http://www.alistapart.com/articles/popup-links/>. Pour vraiment se blinder, on vérifiera que la fenêtre a bien été ouverte (un bloqueur de *pop-ups* un peu trop zélé pourrait l'en avoir empêché), comme décrit sur <http://cookiecrook.com/AIR/2003/train/xmp/popup/pop.js>.
- Ne modifiez le focus *via* un script qu'avec parcimonie. En effet, changer le focus sans intervention de l'utilisateur peut se révéler extrêmement gênant pour ceux utilisant une loupe d'écran voire un lecteur d'écran. C'est donc à éviter, tout particulièrement si vous pensiez le faire périodiquement (par exemple, toutes les 30 secondes) ! En revanche, c'est parfaitement acceptable suite à une action manuelle de l'utilisateur (par exemple, en réaction à l'activation du bouton Lire mes courriels, on peut déplacer le focus sur la liste des courriels après avoir chargée celle-ci).
- Ne limitez pas vos événements traités à ceux spécifiques à la souris, sauf peut-être pour `click`, simulé par tous les navigateurs avec la touche Entrée. Préférez le couplage de la version souris avec la version clavier, par exemple `mouseover` avec `focus` et `mouseout` avec `blur`, etc.

Besoins fréquents et solutions concrètes

Armés de toutes ces connaissances nouvelles, nous allons à présent les mettre en application au travers de quelques exemples concrets, qui répondent par ailleurs à des besoins récurrents. Nous ne détaillerons pas chaque script, il s'agit plutôt de faire une démonstration générale. Vous aurez recours à la spécification pour éclaircir les éventuels détails que vous ne saisiriez pas bien. Tous ces exemples figurent dans l'archive des codes source disponible *via* la page de l'ouvrage sur le site des éditions Eyrolles.

Précisons aussi, avant de commencer, que ces exemples seraient souvent considérablement simplifiés par l'emploi judicieux des améliorations fournies par Prototype, mais ne mettons pas la charrue avant les bœufs...

Décoration automatique de libellés

Commençons par une décoration automatique de libellés. Comme vous le savez, l'élément `label` sert à identifier le libellé d'un champ de formulaire. On peut l'associer au champ soit en affectant l'ID du champ à l'attribut `for` du libellé, soit en plaçant le champ dans l'élément `label`, après (ou avant) le texte du libellé.

Nous allons gérer une décoration automatique (au chargement de la page) de ces libellés centrée sur deux aspects :

- 1 Tout libellé disposant d'un attribut `accesskey` tentera de souligner la lettre correspondante dans son texte, pour rendre la touche de raccourci évidente visuellement.
- 2 Tout libellé disposant d'un attribut `for` (méthode préférée d'association) vérifiera que le champ existe, et si tel est le cas, examinera l'ID du champ pour y détecter le texte 'Req' : si ce texte est présent, il considérera que le champ est requis, et s'ajoutera donc la classe CSS `required`, que nous aurons définie comme affichant le texte en gras (et sur les navigateurs supportant la pseudoclasse `:after` et la propriété `content`, nous ajouterons une astérisque dynamique après le libellé).

Voici le fichier `index.html` sur lequel nous allons appliquer notre script :

Listing 3-10 Le HTML qui va subir notre décoration automatique

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ><head>
    <meta http-equiv="Content-Type" content="text/html;
      > charset=iso-8859-15" />
    <title>Exemple DOM n°1 : décoration automatique de labels
      ></title>
    <link rel="stylesheet" type="text/css" href="demo.css" />
    <script type="text/javascript" src="demo.js"></script>
  </head>
  <body>
    <h1>Décoration automatique de libellés</h1>
    <p>Examinez le code source du formulaire
      ci-dessous, et comparez à ce que vous obtenez visuellement.</p>
```

```
<p>(Notez que l'envoi du formulaire ne mènera à rien de particulier)</p>

<form id="demoForm" method="get" action="http://www.example.com">
<p>
    <label for="edtReqLogin" accesskey="D">Identifiant</label>
    <input type="text" id="edtReqLogin" name="login" tabindex="1" />
</p>
<p>
    <label for="edtFirstName" accesskey="P">Prénom</label>
    <input type="text" id="edtFirstName" name="firstName" tabindex="2" />
</p>
<p>
    <label for="edtLastName" accesskey="N">Nom</label>
    <input type="text" id="edtLastName" name="lastName" tabindex="3" />
</p>
<p class="submit">
    <input type="submit" value="Envoyer" accesskey="E" tabindex="4" />
</p>
</form>

</body>
</html>
```

Notez l'absence de tout balisage non sémantique : ni `table` ni `fatras` de `div` et `span` pour la mise en page. Et pourtant, on pourra obtenir un aspect irréprochable grâce à la feuille de styles, dont voici le contenu (`demo.css`) :

Listing 3-11 Notre feuille de styles pour cet exemple

```
form#demoForm {
    width: 40ex;
    padding: 1em;
    margin: 2em auto;
    border: 1ex solid silver;
    background: #eee;
    font-family: sans-serif;
}

form#demoForm p {
    position: relative;
    margin: 0 0 0.5em;
}

form#demoForm p.submit {
    margin: 0;
    text-align: right;
}
```

```

input#edtReqLogin, input#edtFirstName, input#edtLastName {
    position: absolute;
    left: 20ex;
    right: 0;
}

input:focus {
    border: 2px solid black;
    background: #ffd;
}

label.required {
    font-weight: bold;
}

label.required:after {
    content: '*';
}

span.accessKey {
    text-decoration: underline;
}

```

Enfin, voici notre script `demo.js`, qui constitue le sel de l'exemple :

Listing 3-12 Notre script de décoration automatique des libellés

```

// Être compatible avec MSIE...
if ('undefined' == typeof Node)
    Node = { ELEMENT_NODE: 1, TEXT_NODE: 3 };

function addListener(element, baseName, handler) {
    if (element.addEventListener)
        element.addEventListener(baseName, handler, false);
    else if (element.attachEvent)
        element.attachEvent('on' + baseName, handler);
} // addListener

function decorateLabels() {
    var labels = document.getElementsByTagName('label');
    var label, len = labels.length;
    for (var index = 0; index < len; ++index) {
        label = labels[index];
        if (label.accessKey) {
            var ak = label.accessKey.toUpperCase();
            decorateNodeForAccessKey(label, ak);
        }
    }
}

```

```
        if (label.htmlFor) {
            var elt = document.getElementById(label.htmlFor);
            if (!elt)
                continue;
            if (elt.id.match(/Req/))
                label.className += ' required';
        }
    }
} // decorateLabels

function decorateNodeForAccessKey(elt, key) {
    if (Node.ELEMENT_NODE == elt.nodeType) {
        var node = elt.firstChild;
        while (node && !decorateNodeForAccessKey(node, key))
            node = node.nextSibling;
        // Si node n'est pas null, on a trouvé l'AK dans un descendant
        // et on a décoré : on renvoie non-null, équivalent à true
        return node;
    }
    if (Node.TEXT_NODE != elt.nodeType)
        return false;
    var pos = elt.nodeValue.toUpperCase().indexOf(key);
    if (-1 == pos)
        return false;
    var suffix = elt.nodeValue.substring(pos + 1);
    var akSpan = document.createElement('span');
    akSpan.className = 'accessKey';
    akSpan.appendChild(document.createTextNode(elt.nodeValue.charAt(pos)));
    // On évite node.splitText et node.deleteData sur MSIE...
    // On manipulenodeValue et on crée le deuxième nœud Texte manuellement.
    elt.nodeValue = elt.nodeValue.substring(0, pos);
    elt.parentNode.appendChild(akSpan);
    elt.parentNode.appendChild(document.createTextNode(suffix));
    // Très important pour éviter une récursion infinie !
    return true;
} // decorateNodeForAccessKey

addListener(window, 'load', decorateLabels);
```

Voici notre page après chargement.

Figure 3-10

Notre page chargée, avec ses libellés décorés



C'est déjà un bel exemple d'application, plutôt complet. Je vous encourage à l'améliorer au travers de deux petits exercices.

- 1 Listez toutes les valeurs de l'attribut `accesskey` (pas seulement dans les libellés, mais dans tous les éléments : utilisez `getElementsByName('*')`) et détectez les collisions : ajoutez alors un cadre rouge aux libellés à l'aide de leur propriété `style:label.style.border = '2px solid red'`; (ou mieux, hors MSIE, utilisez `outline` plutôt que `border`). Vous vous rendrez ainsi un fier service, car on a vite fait d'associer deux fois le même raccourci au fil de l'évolution de la page.
- 2 Toujours en itérant sur tous les éléments dotés d'un attribut `accesskey`, ajustez l'attribut `title` des éléments bénéficiant du raccourci (pour un libellé, il faut aller sur l'élément référencé dans `for`), soit en lui ajoutant un texte de type '(Alt+X)' si une valeur existe déjà, soit en créant la valeur 'Alt+X'. Si vous voulez pousser, vous pourrez même détecter que vous êtes sur Mac OS, en cherchant par exemple le texte 'Macintosh' à l'intérieur de `navigator.userAgent` et utiliser 'Cmd' plutôt que 'Alt'...

Validation automatique de formulaires

Toujours plus fort, nous allons maintenant fournir une validation avancée de formulaires. Ce chapitre se concentrant sur le DOM, je ne vous fournis plus à présent que le script. Vous trouverez la démonstration complète dans l'archive des codes source disponible sur le site des éditions Eyrolles.

Qu'entendons-nous par « validation automatique » ?

- 1 Chaque formulaire du document se voit associer notre gestionnaire d'interception de l'événement `submit` (qui vient en plus des autres gestionnaires éventuellement enregistrés).
- 2 Le `submit` est donc intercepté par notre gestionnaire, qui récupère tous les champs (`input`, `select`, `textarea`) du formulaire et examine leur ID. Nous prenons en charge une syntaxe particulière dans les ID, décrite plus bas, qui permet de spécifier les contraintes de validation.
- 3 On accumule au fur et à mesure le texte des messages d'erreur dans un unique message. On garde également une référence vers le premier champ fautif.
- 4 En fin de traitement, si aucune erreur n'a été détectée, on ne fait rien, ce qui laisse passer l'envoi du formulaire. En revanche, si on a détecté un pépin, on signale l'erreur (ici avec un `alert`, mais vous pourriez avoir un `div` exprès pour ça, créer une liste `ul/li` de toute pièce, etc.), on met le focus sur le premier champ, on stoppe la propagation et on annule le traitement par défaut.

La syntaxe que nous prendrons en charge pour les ID est la suivante :

`préfixeQue1conque_[Req][_(Int|Db1|Date)][_min[_max]]`

Quelques détails d'interprétation :

- `Req` indique, comme tout à l'heure, que le champ est requis.
- `Int` indique un champ à valeur entière, `Db1` un champ `Double`, donc à virgule flottante, et `Date` un champ date, pour lequel on exigera ici, par simplicité, un format `jj/mm/aaaa`. On ne validera pas la date en profondeur (libre à vous...).
- Pas de `max` sans `min` d'abord. On ne les prend pas en charge pour les dates non plus, par souci de simplicité de l'exemple.

Voici le fragment de HTML qui contient notre formulaire :

Listing 3-13 Notre formulaire et ses ID spécialement conçus

```
<form id="demoForm" method="get" action="http://www.example.com">
<p>
    <label for="edtLogin_Req" accesskey="D">Identifiant</label>
    <input type="text" id="edtLogin_Req" name="login" tabindex="1" />
</p>
```

```

<p>
    <label for="edtAge_Req_Int_1_120" accesskey="A">Age</label>
    <input type="text" id="edtAge_Req_Int_1_120" name="age"
           ↪ tabindex="2" value="toto" />
</p>
<p>
    <label for="edtEuroRate_Db1_0.01" accesskey="T">Taux de l'euro
    </label>
    <input type="text" id="edtEuroRate_Db1_0.01" name="euroRate"
           ↪ tabindex="3" value="6.55957" />
</p>
<p>
    <label for="edtBirthDate_Date" accesskey="N">Date de naissance
    </label>
    <input type="text" id="edtBirthDate_Date" name="birthDate"
           ↪ tabindex="3" value="04/11/1977" />
</p>
<p class="submit">
    <input type="submit" value="Envoyer" accesskey="E" tabindex="4" />
</p>
</form>

```

Et voici les fragments importants du script (le reste est sur le site...) :

Listing 3-14 Notre validation automatique de formulaires

```

REGEX_AUTO_FIELD = /^[^_]+(_Req)?(_(Int|Db1|Date)(_[0-9.]+){0,2})?$/;
REGEX_BLANK = /^\s*$/;
REGEX_DAY = /^(0?[1-9]|1[1-2][0-9]|3[01])$/;
REGEX_MONTH = /^(0?[1-9]|1[0-2])$/;
// Les multiples groupes vont nous découper l'ID tout seuls...
REGEX_TYPED_FIELD = /(_(Int|Db1|Date)(_[0-9.]+))?(_(_[0-9.]+))?$/;
REGEX_YEAR = /^[0-9]{2,4}$/;

...
function addFormChecks() {
    var forms = document.forms, len = forms.length, form;
    for (var index = 0; index < len; ++index) {
        form = forms[index];
        addListener(form, 'submit', checkForm);
    }
} // addFormChecks

...
function checkForm(e) {
    var form = e.target || e.srcElement;
    var field, len = form.elements.length;
    var errors = '';
    var faulty = null;

```

```
for (var index = 0; index < len; ++index) {  
    field = form.elements[index];  
    // Vérification de syntaxe  
    if (!field.id.match(REGEX_AUTO_FIELD))  
        continue;  
    var value = getFieldValue(field);  
    // Champ requis ?  
    if (field.id.match(/_Req/) && value.match(REGEX_BLANK)) {  
        errors += getFieldName(field) + MSG_BLANK + '\n';  
        faulty = faulty || field;  
        continue;  
    }  
    // Champ typé ?  
    var match = field.id.match(REGEX_TYPED_FIELD);  
    if (match) {  
        var type = match[1];  
        var min = match[3];  
        var max = match[5];  
        var error = checkTypedField(value, type, min, max);  
        if (error) {  
            errors += getFieldName(field) + error + '\n';  
            faulty = faulty || field;  
        }  
    }  
}  
if (!faulty)  
    return;  
stopEvent(e);  
alert(errors);  
faulty.focus();  
} // checkForm  
  
function checkTypedField(value, type, min, max) {  
    // Valeurs par défaut pour les bornes  
    min = 0 === min ? min : min || Number.NEGATIVE_INFINITY;  
    max = 0 === max ? max : max || Number.POSITIVE_INFINITY;  
    var val;  
    if ('Int' == type) {  
        try {  
            val = parseInt(value, 10);  
            if (String(val) != value)  
                throw val;  
        } catch (e) {  
            return MSG_NOT_AN_INTEGER;  
        }  
    }  
}
```

```

if ('Dbl' == type) {
    try {
        val = parseFloat(value);
        if (String(val) != value)
            throw val;
    } catch (e) {
        return MSG_NOT_A_DOUBLE;
    }
}
if ('Int' == type || 'Dbl' == type) {
    if (val < min)
        return MSG_TOO_LOW;
    if (val > max)
        return MSG_TOO_HIGH;
}
if ('Date' == type) {
    var comps = value.split('/');
    if (3 != comps.length || !comps[0].match(REGEX_DAY) ||
        !comps[1].match(REGEX_MONTH) ||
        !comps[2].match(REGEX_YEAR))
        return MSG_NOT_A_DATE;
}
return null;
} // checkTypedField
...

```

Les fonctions utilitaires (`getFieldName`, `getFieldValue`, `stopEvent`) et les textes des messages sont laissés de côté, car ils n'apportent pas grand-chose à la fonctionnalité pure de l'exemple. Vous les trouverez dans l'archive disponible en ligne. On a déjà un bon script d'exemple, qui illustre la majorité des éléments techniques vus dans ce chapitre et le précédent !

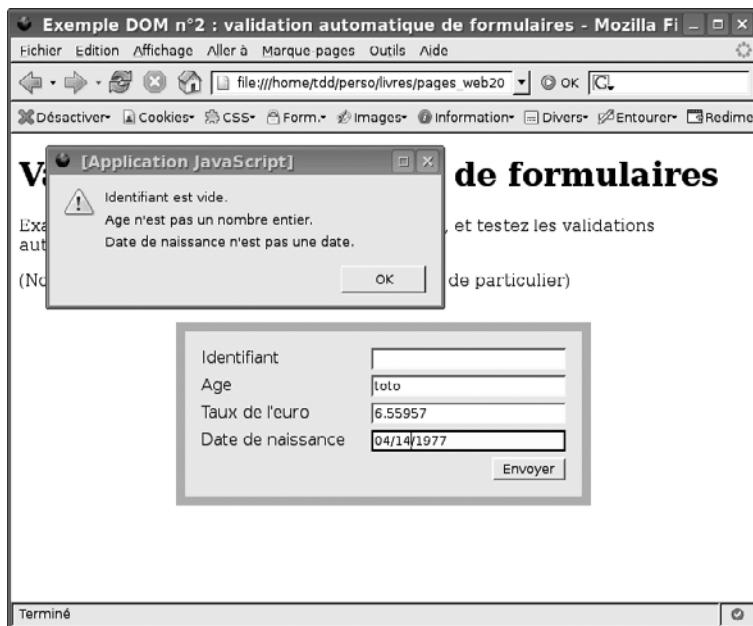
Je ne saurais trop vous recommander de vous faire la main à l'aide des exercices suivants :

- 1 Ajoutez un gestionnaire au chargement qui donne le focus à l'élément de `tabindex` valant 1 (un) dans la page.
- 2 Augmentez la richesse de la syntaxe, en permettant en cas de type `Date` de définir, avant les bornes mais séparé de '`Date`' par un trait de soulignement (`_`), le format de date, parmi les possibilités suivantes : `dmy`, `mdy` et `ymd`. Vous pourrez alors ajuster la validation. Encore mieux : faites qu'un `y` (minuscule) signifie « année sur deux chiffres » et un `Y` (majuscule) « année sur quatre chiffres ».

Vous pouvez voir un exemple d'exécution à la figure 3-11.

Figure 3-11

Exemple de validation automatique



Résoudre les écueils classiques

Ce dernier exemple, en particulier dans sa version complète, disponible en ligne, illustre les principaux écueils du scripting DOM, qui tournent généralement autour des différences entre MSIE et les autres navigateurs.

Il existe aussi, bien sûr, des problèmes plus pointus, présents sur d'autres navigateurs répandus, comme des soucis de positionnement complexe ou de noms d'événements sur Firefox, Safari, etc. Mais il ne s'agit pas d'écueils classiques.

MSIE et la gestion événementielle

On l'a vu dans la section dédiée à ce sujet, quand il s'agit d'événements, MSIE ne fait rien comme les autres : pas de `stopPropagation()`, pas de `preventDefault()`, pas d'interface `Event`, de propriété `target`...

Vous avez pu voir des contournements manuels à l'œuvre dans ce dernier script d'exemple, mais la solution la plus portable consiste à utiliser Prototype et son objet global `Event`. En plus d'offrir des méthodes équivalentes universelles (`stop`, `element`, etc.), il fournit une interface unifiée sur l'ensemble des propriétés spécifiques des événements (état des modificateurs clavier, position de la souris et j'en passe).

MSIE et le DOM de select/option

Il existe également un problème sur MSIE concernant le DOM des éléments `select` et `option`, qui décrivent les champs de type liste. Le DOM prévoit deux comportements :

- une propriété `value` directement au niveau du `select`, qui fournit la valeur actuellement sélectionnée s'il y en a une (la première en cas de sélection multiple) ;
- une propriété `value` au niveau des `options`, qui prend automatiquement la valeur de la propriété `text` si l'attribut `value` n'est pas défini pour la balise `option` correspondante (ce comportement est en fait défini carrément par HTML 4.01).

MSIE ne respecte aucun de ces deux comportements. C'est pourquoi il faut récupérer la valeur en passant par la collection `options` du `select`, sa propriété `selectedIndex`, et les propriétés `value` et `text` de l'`option`. Cet algorithme alourdi est présent dans notre fonction `getFieldValue`, non imprimée ici.

Une solution portable consiste à utiliser la fonction `$F` de Prototype pour récupérer la valeur d'un élément. Elle gérera même correctement les valeurs d'une liste à sélection multiple !

Les principaux points problématiques

Si on fait le bilan de ce que nous avons appris sur le DOM, on constate qu'on a à portée de main une puissance de traitement extraordinaire, avec la possibilité d'explorer les moindres recoins du document et de le manipuler comme bon nous semble.

Hélas ! Le prix de ces possibilités est un peu lourd à payer :

- La mauvaise prise en charge du DOM niveau 2 (ou même niveau 1, pour certains points) par MSIE pose de graves problèmes de portabilité, nous forçant à alourdir considérablement notre code.
- D'autres navigateurs ont aussi quelques accrocs dans leur prise en charge du DOM, bien que sur des points beaucoup plus bénins.
- Les codes créant des fragments de DOM sont généralement verbeux : les interfaces qui nous sont fournies sont puissantes, mais il faut beaucoup de code pour créer quoi que ce soit, même des fragments très simples, du type :
`` ou `<h1>Merci !</h1>`.
- Il est impossible de réaliser une extraction sur la base d'une règle CSS, alors que c'est la syntaxe de sélection que connaissent le mieux les développeurs web.

Toutefois, des solutions existent, même si elles ne sont pas forcément fournies par un standard W3C.

Ainsi, Prototype simplifie beaucoup de choses et ajoute de nombreuses possibilités. Nous allons nous régaler au chapitre suivant, en découvrant toute cette puissance conçue pour être facile d'emploi !

La bibliothèque script.aculo.us, basée sur Prototype, fournit un objet `Builder` conçu spécialement pour simplifier la création de fragments DOM. Nous ne l'étudierons pas dans le chapitre 8, qui couvre la majorité de script.aculo.us en détail, mais des exemples clairs sont disponibles sur le site de la bibliothèque.

Pour aller plus loin

Livres

DOM Scripting

Jeremy Keith

Friends of ED, septembre 2005, 341 pages

ISBN 1-590-59533-5

Sites

- DOM Scripting, le site :
<http://domscripting.com>
- Le site de Jeremy Keith, gourou du scripting DOM :
<http://adactio.com>
- Le site de Peter-Paul Koch, autre gourou du sujet :
<http://www.quirksmode.org>
- La Mozilla Developer Connection a une référence complète de JavaScript et des informations sur le « DOM navigateur » (objets `navigator`, `window`, etc.) :
 - <http://developer.mozilla.org/en/docs/JavaScript>
 - <http://developer.mozilla.org/fr/docs/JavaScript>
 - <http://developer.mozilla.org/fr/docs/DOM>
- Les spécifications sont le point de référence incontournable !
<http://www.w3.org/DOM/DOMTR>
- Le Web Standards Project (WaSP) fait avancer les standards qui comptent et sa DOM Scripting Task Force abat un boulot titanique :
<http://webstandards.org/action/dstf>

4

Prototype : simple, pratique, élégant, portable !

Prototype est une bibliothèque JavaScript qui simplifie énormément la majorité des utilisations courantes de JavaScript, même lorsqu'il s'agit de fonctions avancées. Elle fait figure de précurseur (remontant à début 2005) et a été initiée par Sam Stephenson. Aujourd'hui encore, en dépit de la progression rapide d'alternatives comme jQuery, Prototype reste le *framework* JavaScript le plus utilisé sur le Web.

Très bien structurée et organisée avec beaucoup de cohérence, Prototype dope notamment sérieusement les classes incontournables (`Array`, `String`, `Number`, `Function`) et les éléments HTML. Tout ceci en assurant une excellente portabilité d'un navigateur à l'autre, ce qui constitue d'ordinaire une épine permanente dans le pied des développeurs web.

Ce qui est certain, c'est que les débutants JavaScript comme les gourous expérimentés prendront tout à coup beaucoup plus de plaisir à écrire du JavaScript s'ils utilisent Prototype. Alors vous aussi, faites-vous plaisir ; il vous suffit de lire ce chapitre.

Avant de commencer...

Prototype propose quelques classes tout à fait remarquables pour encapsuler les traitements Ajax, mais nous attendrons le chapitre 7 pour les voir, après avoir mis les mains dans XMLHttpRequest. Les détailler ici soulèverait probablement trop de questions, alors qu'elles trouvent naturellement leur place à la suite d'un examen approfondi des détails qu'elles simplifient.

Sachez également que la plupart des exemples de plus de 3 lignes de ce chapitre sont fournis avec leurs pages de test dans l'archive de codes source pour ce livre, disponible sur le site des Éditions Eyrolles.

Enfin, je me dois de signaler un point important par rapport à la première édition : trois mois après la sortie de cette dernière, j'ai rejoint Prototype Core, la petite équipe de développeurs qui conçoivent et implémentent Prototype. Certaines fonctionnalités décrites dans ce chapitre sont le fruit de mon travail... Après 18 mois de collaboration sur ce projet, je confesse donc une légère partialité et, à l'évidence, un enthousiasme débordant.

Un mot sur les versions

La première édition de ce livre documentait, avec une avance significative, Prototype 1.5 RC1. Il s'agissait d'ailleurs du premier véritable effort de documentation de Prototype !

En deux ans, la bibliothèque a parcouru du chemin au travers des versions 1.5 finale, 1.5.1 (qui a tellement optimisé l'incontournable fonction \$\$ qu'elle a levé toutes les contraintes à son usage), 1.5.1.1 et 1.6.0 (qui a notamment proposé deux vrais systèmes de classes et d'événements DOM personnalisés). Ces derniers temps, on a vu la sortie des versions 1.6.0.1 à 1.6.0.3 (centrées sur des correctifs et optimisations), et la 1.6.1 devrait apporter quelques nouveautés alléchantes. Tout ceci en attendant la 2.0, dont la date de sortie, même très approximative, n'est pas encore fixée.

Ce chapitre documente la version 1.6.0.3, sur la base de son code source actuel dans GitHub, quelques jours avant sa sortie définitive. Quand bien même la version 1.6.1 verrait le jour très rapidement, vous êtes ainsi à jour, ou peu s'en faut.

Pour obtenir Prototype, une seule adresse : le site officiel prototypejs.org. Le téléchargement se fait sur <http://prototypejs.org/download>. Vous pouvez d'ailleurs aussi y récupérer de plus anciennes versions en cas de besoin (jusqu'à la 1.5.0 du 18 janvier 2007).

Ce site officiel est d'ailleurs précieux, puisqu'on y trouve notamment :

- la documentation de référence : <http://prototypejs.org/api> ;

- quelques didacticiels sur des sujets clefs tels que le système de classes, les possibilités Ajax ou l'extension du DOM : <http://prototypejs.org/learn> ;
- l'actualité de Prototype : <http://prototypejs.org/blog>.

Navigateurs pris en charge

Prototype 1.6.0.3 passe l'ensemble de ses tests (environ 350 tests unitaires et plus d'une centaine de tests fonctionnels) sur les navigateurs suivants :

- MSIE 6+ ;
- Firefox 1.5+ ;
- Safari 2+ ;
- Opera 9.25+.

Il s'agit tout de même de plus de 98 % des navigateurs déployés à ce jour... Par extension, de nombreux navigateurs basés sur les mêmes moteurs ne présentent aucun problème, par exemple les versions récentes de Camino, Konqueror, Mozilla, etc.

Vocabulaire et concepts

Commençons par rappeler qu'en JavaScript, il n'y a pas à proprement parler de classes : tout est objet. L'héritage est obtenu par ajout de champs et de méthodes au prototype d'un objet. Nous avons couvert cela au chapitre 2.

Espaces de noms et modules

J'aurai souvent recours aux termes « espace de noms » et « module », que j'utilise ici de façon presque interchangeable. Lorsque je qualifie un objet d'**espace de noms**, je veux dire qu'il n'est pas destiné à être instancié, ni même incorporé au prototype d'un autre objet. Il n'existe que pour donner un contexte nommé à ses méthodes et leur éviter ainsi d'être globales, mais aussi d'avoir à utiliser un nom plus long.

On trouve ainsi par exemple les objets Ajax, Prototype et Abstract, qui ne constituent que des espaces de noms, comme un *namespace* C++, un paquet Java ou une unité Delphi.

Je qualifie par ailleurs certains objets de **modules**. Ces objets servent à regrouper des méthodes autour d'un aspect particulier, lesdites méthodes étant vouées à être incorporées aux prototypes d'autres objets afin de leur ajouter cet aspect. J'emploie ici le terme « aspect » au sens de la *programmation orientée aspects* (AOP). Le terme « module » prend ici le sens exact qu'il a, par exemple, en Ruby.

Prototype fournit plusieurs modules, dont Enumerable et Element.Methods.

Itérateurs

Vous allez aussi rencontrer le terme « itérateur » à tout bout de champ. Dans le cadre de Prototype, je qualifierai d'**itérateur** une fonction destinée à être invoquée sur chaque élément d'une itération. Cette itération est réalisée par une fonction qui reçoit la nôtre en argument. Ainsi, `Prototype.K` est exclusivement utilisée, en interne, comme « itérateur par défaut ». Il s'agit là d'une légère différence avec le sens classique du mot, qui désigne le mécanisme d'itération, et non celui d'opération sur les éléments produits par l'itération.

Les itérateurs permettent de séparer la logique d'itération de celle d'action, qui opère sur chaque élément. Vous les trouverez dans la plupart des bibliothèques standardisées de classe, mais il ne s'agit pas toujours de la fonction opérative, comme ici avec Prototype.

En C++, un itérateur est un objet encapsulant la logique d'itération, et non celle d'opération. Il suffit généralement qu'il fournisse les opérateurs `*`, `->` et `++`. En Java aussi, l'itérateur encapsule l'itération elle-même, sous la forme de l'interface `java.util.Iterator`. En Ruby, un itérateur est une méthode encapsulant l'itération, qui reçoit un bloc ou une `Proc` en argument, auquel la méthode passe tour à tour chaque élément de l'itération. Vous trouverez des équivalents en Perl, Python, C#, etc.

Élément étendu

Prototype estime que les éléments HTML, tels que les fournit le DOM, sont un peu nus. Il est vrai qu'au regard des utilisations communes qu'on en fait, l'interface `HTMLElement` du DOM niveau 2 HTML (voir chapitre 3) est assez légère. Prototype fait donc tout son possible pour enrichir les éléments qu'il vous renvoie.

Si le navigateur offre un mécanisme de prototype pour les éléments du DOM, Prototype enrichit automatiquement tous les éléments des méthodes présentes dans le module `Element.Methods`, que nous verrons en détail plus loin.

Par ailleurs, tout élément de type champ de formulaire reçoit également les méthodes de `Form.Element.Methods` (toutes celles de `Form.Element` moins `focus` et `select`, qui existent déjà en natif). Dans le même esprit, tout élément `form` reçoit également les méthodes de `Form.Methods` (toutes celles de `Form` sauf `reset`, même remarque).

Si cette possibilité lui est refusée (par exemple sur MSIE), Prototype enrichira « à la volée » tout élément accédé au travers de son API, ce qui en pratique signifie que

dans la très vaste majorité des cas, ces méthodes sont en accès direct sur les éléments manipulés. Sur un tel élément `elt`, au lieu de faire :

```
Element.methodeSympa(elt)
Element.autreMethodeSympa(elt, arg1, arg2...)
```

Vous pouvez directement faire :

```
elt.methodeSympa()
elt.autreMethodeSympa(arg1, arg2...)
```

Ce qui est plus sympathique, plus orienté objet, et plus court !

Nous y reviendrons dans la section consacrée à `Element.Methods`. D'ici là, gardez simplement ceci à l'esprit :

- Partout où j'indique qu'une fonction prend en argument un élément (ou plusieurs), vous pouvez passer soit un élément existant, soit son ID : de tels arguments sont toujours utilisés au travers de la fonction `$`, que nous allons voir dans un instant. Ces éléments sont donc étendus quoi qu'il arrive.
- Quand je dis qu'une fonction renvoie un « élément étendu » (ou un tableau de tels éléments), ces éléments sont dotés directement des méthodes du module `Element.Methods` (voire d'autres modules spécifiques à la balise de l'élément).

Alias

Dans un souci de confort maximal, Prototype *alias* parfois certaines méthodes, lorsque celles-ci existent fréquemment sous deux noms différents dans les bibliothèques les plus répandues. Ainsi, `collect` et `map` sont synonymes, de même que `detect` et `find`, `select` et `findAll`, ou encore `include` et `member`. L'objectif est de vous permettre d'utiliser une méthode par un nom qui vous est familier, pour optimiser votre courbe d'apprentissage.

Dans de tels cas :

- je précise les deux noms dans les titres de section ;
- j'indique dans le corps du texte que les méthodes sont des alias ;
- je précise l'alias dans les blocs de syntaxe à l'aide d'une barre oblique (/), comme ceci :

```
objet.nom1/nom2(arguments)
```

Comment utiliser Prototype ?

C'est très simple. Prototype est fourni sous la forme d'un unique fichier de script, `prototype.js`, d'environ 124 Ko avant toute forme de compression (moins d'une seconde de téléchargement, et le cache de votre navigateur le gardera bien au chaud par la suite ; on peut aussi configurer le serveur web pour servir ces fichiers *gzippés*, ici à environ 30 Ko, ou encore mieux : tirer parti du réseau de distribution Google¹). Il vous suffit de charger ce script depuis l'en-tête de votre page (X)HTML, de préférence *avant* les autres scripts, ces derniers ayant tout intérêt à s'en servir :

```
<head>
...
<script type="text/javascript" src=".../prototype.js"></script>
...
</head>
```

C'est tout !

Enfin, sachez que l'archive de codes source pour ce livre, disponible sur le site des Éditions Eyrolles, contient de nombreuses pages complètes d'exemple pour l'ensemble des fonctionnalités vues dans ce chapitre. Je vous invite à la télécharger et à la décompresser, si ce n'est déjà fait, pour pouvoir découvrir dans le détail chaque objet en action, en plus des exemples succincts contenus dans le texte.

Vous allez aimer les dollars

Loin de moi l'idée de vous taxer de mercantilisme, mais il se trouve que Prototype a choisi, pour des raisons de confort, de fournir ses fonctions les plus utiles (que vous trouverez vite indispensables !) sous des noms les plus courts possible. Le meilleur moyen pour éviter les conflits avec des noms existants était l'emploi du préfixe \$, dont on oublie trop souvent qu'il est autorisé comme caractère d'identifiant en JavaScript (ECMA-262, 7.6§2).

Il existe donc sept fonctions globales d'intérêt majeur : \$ (oui, juste \$! Ça devient difficile de faire plus court, et donc plus discret, dans votre code !), \$A, \$H, \$F, \$R, \$\$ et \$w. Pour les décrire, je vais parfois devoir faire appel à des concepts qui seront exposés plus loin dans ce chapitre, mais cela ne devrait pas poser de problèmes de compréhension.

1. Voir <http://code.google.com/apis/ajaxlibs/>

La fonction \$ facilite l'accès aux éléments

C'est sans conteste la fonction la plus utilisée de Prototype. Son objectif : vous permettre d'accéder aux éléments du DOM par leur identifiant, et restreindre le plus possible les cas particuliers et les tests. Je vais m'étendre un peu sur sa description, car elle constitue vraiment une pierre angulaire de Prototype, et on l'utilise souvent en deçà de ses capacités.

Voici ses invocations possibles :

```
| $(id) -> objÉtendu  
| $(obj) -> objÉtendu  
| $(id, id, obj...) -> [objÉtendu, objÉtendu, objÉtendu...]
```

À l'avenir, je résumerai ce genre de possibilités comme ceci :

SYNTAXE

```
| $((id/obj)...)>> objÉtendu / [objÉtendu...]
```

Si vous lui passez un identifiant (une `String`), elle récupère l'élément avec un `document.getElementById`. Sinon, elle considère que vous lui passez en réalité l'élément lui-même. Dans tous les cas, elle enrichit l'élément si besoin pour s'assurer que vous récupérez au final un élément étendu.

La plupart des fonctions de Prototype qui acceptent des éléments en argument commencent par les passer à \$, histoire d'accepter tant des ID que des éléments déjà récupérés, et d'être certaines d'utiliser leur version étendue. Je ne préciserais donc plus, à l'avenir, qu'une fonction prend un ID ou un élément : je dirai juste « un élément ».

Si vous ne lui avez passé qu'un élément ou ID, elle vous renvoie l'élément étendu. Si elle a reçu plusieurs arguments, elle renvoie un tableau des éléments étendus correspondants. Si vous souhaitez constituer un tableau d'éléments, il est donc inutile de faire :

```
// Code inutilement complexe !  
var items = []  
items[0] = $('item0');  
items[1] = $('item1');  
items[2] = $('item2');
```

Préférez :

```
// Code bien plus pratique  
var items = $('item0', 'item1', 'item2');
```

La fonction \$A joue sur plusieurs tableaux

Nous allons le voir dans quelques pages, l'objet natif JavaScript Array est très développé par Prototype, qui y incorpore notamment le module Enumerable. Un tableau est donc bien plus puissant que la plupart des collections renvoyées par le DOM (notamment les objets NodeList et HTMLCollection), qui n'offrent pour la plupart que les propriétés `length` et `item`.

C'est pourquoi la fonction \$A est importante, et souvent utilisée. Elle prend un argument susceptible d'être transformé en tableau, et en fait un objet Array digne de ce nom, doté de toutes les extensions dues à Enumerable !

SYNTAXE

\$A(obj) -> tableau

Reste à savoir ce qui constitue un argument *susceptible d'être transformé en tableau*. \$A fonctionne comme ceci :

- 1 Si on lui passe `null`, `undefined` ou un objet qui ne correspond pas aux deux cas qui suivent, elle renvoie un tableau vide.
- 2 Si l'objet reçu implémente la méthode `toArray()`, \$A utilise cette méthode. Cela permet à nos classes de contrôler leur transformation en tableau, si nécessaire. C'est notamment le cas de tout objet incorporant le module Enumerable.
- 3 Sinon, l'objet passé doit disposer d'une propriété `length` et de propriétés d'indice numérique, afin de pouvoir être indexé comme un tableau. Chaque propriété ainsi accédée sera ajoutée au tableau résultat.

Voici un exemple complet de tous les cas d'utilisation :

Listing 4.1 \$A dans tous ses états !

```
$A(null)
// => []
$A()
// => []
$A({ niceTry: 'mais non' })
// => []
$A([])
// => []
$A(['et', 'hop', 'facile !'])
// => ['et', 'hop', 'facile !']
```

```
var buddy = {
  firstName: 'Amir',
  lastName: 'Jaballah',
  toArray: function() {
    return [this.firstName, this.lastName];
  }
};

$A(buddy)
// => ['Amir', 'Jaballah']

var convoluted = {
  0: 'eh bien',
  1: 'voilà un exemple',
  2: 'pour le moins tordu !',
  length: 3
};

$A(convoluted)
// => ['eh bien', 'voilà un exemple', 'pour le moins tordu !']
```

La fonction \$H, pour créer un Hash

Nous verrons plus loin que Prototype définit un objet sympathique, `Hash`, qui représente un tableau associatif (un peu comme `java.util.HashMap` ou les tableaux de PHP, par exemple). Si vous avez lu attentivement le chapitre 2, vous savez qu'un objet JavaScript est, essentiellement, un tableau associatif dont les clefs sont les noms des propriétés, tandis que les valeurs sont... eh bien, les valeurs des propriétés. Au premier abord, `Hash` n'ajoute donc rien à la sauce.

Nous verrons qu'en réalité, si. Et c'est un type de données si fréquemment utilisé qu'il dispose de sa propre fonction de conversion, `$H`. En réalité, l'un ne va pas sans l'autre : il n'existe pas d'autre moyen que cette fonction pour obtenir une instance de `Hash`.

SYNTAXE

```
$H([objet]) -> Hash
```

On peut donc créer un `Hash` vierge simplement avec `$H()`, ou obtenir la version `Hash` d'un objet existant pour l'examiner plus confortablement, en faisant `$H(objet)`. La fonction s'assure par ailleurs que l'objet résultat incorpore le module `Enumerable`, ce qui ajoute encore aux possibilités.

La fonction **\$F**, des valeurs qui sont les vôtres

Si vous avez déjà essayé de récupérer de façon générique la valeur d'un champ, vous savez que ce n'est pas si trivial. La plupart des types de champ fournissent une propriété `value`, mais on ne doit pas la prendre en compte pour des cases à cocher et boutons radio décochés, et `select` soulève plusieurs questions, suivant qu'il autorise ou non une sélection multiple, ou qu'on est sur MSIE et que la propriété `value` de ses options est mal implémentée. Dans le dernier exemple du chapitre précédent, le code source de la fonction `getFieldValue` illustre bien cette complexité.

Avec Prototype, on fait `$F(élément)`. C'est tout. Je vous le remets comme code individuel, pour vous faciliter la lecture en diagonale à l'avenir :

SYNTAXE

```
$F(element) -> valeurEnTexte / [valeurEnTexte...]
```

Petite précision toutefois : si l'élément indiqué est un `select` à sélection multiple, on récupère un *tableau* des valeurs pour les options sélectionnées.

La fonction **\$R** et les intervalles

Un des objets sous-utilisés de Prototype est `ObjectRange`, qui permet de représenter un intervalle de valeurs pour n'importe quel objet, du moment que celui-ci fournit une méthode `succ()`. La manière la plus simple de créer un `ObjectRange` est d'utiliser la fonction `$R`. C'est très utile pour séparer une définition de boucle de son utilisation. `ObjectRange` inclut par ailleurs `Enumerable`, et hérite donc de sa méthode `toArray()`.

Nous reviendrons sur `$R` dans la documentation d'`ObjectRange`, mais voici tout de même sa syntaxe :

SYNTAXE

```
$R(debut, fin[, finExclude = false]) -> ObjectRange
```

L'objet `ObjectRange` renvoyé représente une itération entre `debut` et `fin`, laquelle peut être exclue si vous précisez un troisième argument `true`. Exemple :

```
var loop = $R(1, 42);
...
loop.each(function(i) { // appelée pour i de 1 à 42 } )

$A($R(1, 5))
// => [1, 2, 3, 4, 5]
```

La fonction \$\$ et les sélecteurs CSS

Apparue avec Prototype 1.5.0 et fortement optimisée en 1.5.1, \$\$ permet de **récupérer tous les éléments correspondant à des sélecteurs CSS**. Indépendante du moteur CSS de votre navigateur, elle prend en charge presque tous les sélecteurs de CSS 3 ! Vous n'imaginez pas combien on s'en sert... Seule \$ est plus populaire. \$\$ est en fait une fonction d'enrobage autour de la classe Selector qui représente, comme son nom l'indique, un sélecteur CSS. La syntaxe est la suivante :

SYNTAXE

```
|| $$ (sélecteur...) -> [objÉtendu...]
```

On récupère un tableau de tous les éléments correspondant aux divers sélecteurs, sans doublons et dans l'ordre du document. Les éléments retournés sont garantis étendus, comme pour \$.

Nous reviendrons dans la section dédiée à Selector sur les syntaxes possibles.

Voici un exemple de code qui masque tous les paragraphes ayant une classe `toggle` placés immédiatement après un h2 :

```
|| $$ ('h2 + p.toggle').invoke('hide');  
// hide() est disponible car les éléments renvoyés sont étendus
```

La fonction \$w joue sur les mots

Il existe une tâche extrêmement fréquente de création de tableau, c'est celle qui consiste à créer une liste de « mots » ; par exemple, des noms de jours, de mois, de champs de formulaires...

En temps normal, on doit écrire le tableau littéral, quelque chose comme ceci :

```
var fields = ['fname', 'lname', 'email', 'country', 'password'];
```

Ce type de saisie a vite fait de déraper avec l'oublier ou l'ajout d'une apostrophe, d'un guillemet ou d'une virgule. Prototype fournit donc l'équivalent du %w de Ruby et du qw de Perl, sous le nom \$w :

```
var fields = $w('fname lname email country password');
```

\$w découpe tout bêtement la chaîne de caractères qui lui est passée sur la base de caractères d'espacement (espaces, tabulations, retours chariot et sauts de ligne). Du coup, on ne peut s'en servir si l'une des valeurs comprend de l'espacement, mais dans nombre de cas c'est bien pratique.

Jouer sur les itérations avec \$break et return

Nous allons aborder tout au long de ce chapitre de nombreuses méthodes encapsulant des itérations, par exemple `each`, `collect`, `inject` ou `map`. Ces méthodes ont recours à des itérateurs pour traiter tour à tour les objets sur lesquels on boucle.

SYNTAXE

```
throw $break;
return;
```

Dans une boucle classique, JavaScript fournit deux moyens de court-circuit, présents dans de nombreux autres langages et signalés au chapitre 2 : `break` et `continue`. Le premier « casse » la boucle, l'exécution continuant à la première instruction après la boucle. Le second court-circuite juste le tour courant, faisant immédiatement passer la boucle à l'itération suivante.

Lorsqu'on utilise des itérateurs, ces mots réservés de JavaScript ne nous sont plus d'aucune utilité puisque le code réalisant la boucle et celui réalisant le traitement sont dans deux fonctions distinctes : respectivement la méthode d'itération et l'itérateur.

Afin de vous fournir néanmoins ces possibilités de court-circuit, Prototype définit donc une « exception globale » : `$break`. Pour obtenir un résultat équivalent au mot réservé, il vous suffit dans votre itérateur de lancer l'exception correspondante. Quant à `continue`, vu qu'on est maintenant dans une fonction, il suffit de faire un `return` pour obtenir un comportement équivalent.

Voici par exemple un code qui récupère dans une liste les 5 premiers candidats ayant plus de 15 de moyenne, et court-circuite l'itération dès qu'on a atteint ce nombre.

Listing 4.2 Court-circuit dans un itérateur avec \$break

```
// Chaque candidat a des propriétés name et grade, et candidates
// est un gros tableau de candidats.
function get5FirstGoodCandidates(candidates) {
    var count = 0;
    var result = [];
    candidates.each(function (c) {
        if (c.grade >= 15) {
            result.push(c);
            if (5 == ++count)
                throw $break;
        }
    });
    return result;
} // get5FirstGoodCandidates
```

Extensions aux objets existants

J'ai classé les apports de Prototype en deux grandes catégories : d'abord les extensions aux objets existants, ensuite les ajouts purs et simples. Nous allons voir les extensions d'abord, parce que ce sont elles qui auront sans doute le plus grand impact sur votre code. En effet, il s'agit d'objets dont vous croyez connaître toutes les possibilités, car ils vous sont familiers. Le risque de sous-exploiter leurs nouvelles capacités est donc élevé. Et ce serait vraiment dommage...

Un Object introspectif

Object est un espace de noms qui fournit nombre de méthodes orientées vers l'examen d'objets, que ce soit dans leur nature ou dans leur contenu.

```
Object.clone(objet) -> cloneDeObjet
Object.extend(destination, source) -> destinationModifiée
Object.inspect(objet) -> représentationString
Object.isArray(objet) -> booléen
Object.isElement(objet) -> booléen
ObjectisFunction(objet) -> booléen
Object.isHash(objet) -> booléen
Object.isNumber(objet) -> booléen
Object.isString(objet) -> booléen
Object.isUndefined(objet) -> booléen
Object.keys(objet) -> Enumerable
Object.values(objet) -> Enumerable
```

SYNTAXE

Object est doté d'une petite méthode sympathique nommée `inspect`, destinée aux développeurs web en train de mettre leur code au point. Elle tente de fournir une représentation textuelle la plus efficace possible ; en effet, la représentation par défaut des objets sous forme de `String` laisse souvent à désirer.

Ainsi, les tableaux apparaissent soit sous la forme de leurs valeurs séparées par des virgules (donc un affichage totalement vide pour un tableau vide, ou incompréhensible pour `[' ', '', ' ', '\t\n']`), soit carrément sous la très inutile forme « `[Array]` »... `String` ne fait guère mieux. `undefined` et `null` sont le plus souvent invisibles. Quant aux objets, n'en parlons pas.

Voici comment procède `Object.inspect` :

- `null` et `undefined` donnent '`null`' et '`undefined`'. Enfin, en théorie, parce qu'à l'heure actuelle, l'utilisation malheureuse de `==` au lieu de `==` fait que les deux donnent '`undefined`'...

- Si l'objet dispose d'une méthode `inspect()`, celle-ci est appelée (Prototype en fournit pour `String`, `Enumerable`, `Array`, `Hash`, `Element`, `Selector` et `Event`, ce qui couvre l'essentiel des besoins).
- À défaut, la méthode `toString()` est appelée. Il faut savoir que tout objet JavaScript a par défaut une méthode `toString()`, et ce depuis toujours (JavaScript 1.0).

Comparez quelques exemples de représentation standard avec `toString()` et de résultat avec `inspect()`. Vous allez sentir la différence en termes de débogage :

Listing 4.3 `toString()` vs. `inspect()`

```
var tableau = [ '', 'Salut', 'C\'est génial !', '\n', '"TDD"' ];
var obj = { nom: 'Christophe', age: 28 };
var objH = $H(obj);

alert(tableau + '\n' + obj + '\n' + objH);
alert(Object.inspect(tableau) + '\n' +
      Object.inspect(obj) + '\n' +
      Object.inspect(objH));
```

Voici les résultats :

Figure 4–1
`toString()` vs. `inspect()`



Il y a déjà du mieux... En combinant avec l'objet global `console` fourni par Firebug (voir chapitre 2), on a de quoi faire.

`Object` nous permet par ailleurs de déterminer de façon fiable le type d'une valeur, sans se soucier des éventuelles incompatibilités inter-navigateurs, à l'aide de ses nombreuses méthodes `is...` Leurs noms parlent d'eux-mêmes, quelques petites précisions cependant :

- `isElement` vérifie que l'argument est un élément DOM (un noeud de type `Node.ELEMENT_NODE`).
- `isFunction` se limite aux véritables fonctions et constructeurs, et ne se laissera pas tromper par certains objets issus du DOM qui « ressemblent » à une fonction.

Les méthodes `keys()` et `values()` permettent de traiter un objet comme un *hash* d'associations clef/valeur ; elles renvoient un tableau des noms de propriétés, ou de leurs valeurs, respectivement :

```
Object.keys(tableau) // => [0, 1, 2, 3]
Object.keys(obj) // => [ 'nom', 'age' ]
Object.values(obj) // => [ 'Christophe', 28 ]
```

L'ordre dépend de l'implémentation par le navigateur de la boucle `for...in...` et n'est donc pas garanti d'un navigateur à l'autre.

La méthode `Object.extend` copie toutes les propriétés (méthodes comprises) d'un objet source dans un objet destination, écrasant au passage celles qui existeraient déjà en portant les mêmes noms. Cette petite méthode fondamentale sert de base à des fonctions comme `clone`, le système de classes, la gestion d'options par défaut...

Enfin, la méthode `clone` fournit, comme son nom l'indique, un clone exact de l'objet source.

Gérer correctement le binding

On a vu en détail, au chapitre 2, la question épineuse du *binding*. Il s'agit principalement de pouvoir passer une méthode en argument, pour exécution ultérieure, sans que celle-ci ne perde le lien qui l'attache à un objet particulier.

Prototype fournit deux ajouts à l'objet `Function` qui s'occupent spécifiquement du *binding* : `bind` et `bindAsEventListener`.

SYNTAXE

```
monObjet.methode.bind(monObjet[, arg...])
monObjet.methode.bindAsEventListener(monObjet[, arg...])
```

La méthode `bind` permet de « transformer » une méthode classique en méthode à « *binding* garanti », c'est-à-dire sur laquelle `this` vaudra toujours ce que vous aurez précisé au moment du `bind`. Qui plus est, elle peut lui fournir des arguments pré-remplis, auxquels les arguments d'invocation seront *ajoutés* (pas de remplacement !)

Voyez l'exemple suivant :

Listing 4.4 Avec ou sans bind...

```
var obj = {
  location: 'Paris, France',
  getLocation: function() {
    return this.location;
  } // getLocation
};
```

```

function show(getter) {
    alert(getter());
} // show

show(obj.getLocation);
// => this global (window) : window.location => l'URL de la page
show(obj.getLocation.bind(obj));
// => this sera bien obj : obj.location => « Paris, France »

```

Ce qu'il faut bien comprendre, c'est que `bind` renvoie une nouvelle fonction, appelleable en lieu et place de l'ancienne autant de fois qu'on veut.

On trouve un exemple de `bind` avec un argument prédéfini dans l'étude de cas sur l'API REST Flickr au chapitre 9, pour l'objet de chargement parallèle gXSLT.

Il existe un cas particulier : celui des gestionnaires d'événements. Lorsque vous utilisez des fonctions globales comme `gestionnaire`, vous n'avez rien de spécial à faire. Toutefois, quand vous passez une méthode et que celle-ci a besoin de référencer son instance conteneur avec `this`, il est important de fournir un enrobage approprié de la méthode à l'aide de `bindAsEventListener`. En effet, `bind` ne prendrait pas soin, au cas où vous auriez pré-rempli des arguments, de démarrer la liste d'arguments effectifs avec l'objet événement qui est censé être transmis à tout gestionnaire d'événements ; cet objet serait alors plutôt passé *en dernier*.

Là aussi, un exemple met les choses au clair :

Listing 4.5 Avec ou sans bindAsEventListener...

```

var Howler = {
    prefix: 'HEY ! ',
    report: function(e, info) {
        alert(this.prefix + e.type + ' ' + info);
    }
};

var badReport = Howler.report.bind(Howler, 'KO');
var goodReport = Howler.report.bindAsEventListener(Howler, 'OK');
document.observe('click', badReport);
document.observe('click', goodReport);
// Et un clic n'importe où va donner :
// => 'HEY ! Undefined [object MouseEvent]'
// => 'HEY ! click OK'

```

Que s'est-il passé ici ? Pour le premier gestionnaire, `badReport`, lorsque l'événement `click` s'est déclenché, le navigateur a appelé `badReport(e)`, mais le `bind` qui a créé `badReport` a transformé cela en `Howler.report('KO', e)`, ce qui n'est pas compatible avec notre signature (l'objet événement étant toujours passé en premier en

temps normal, cette signature est pourtant correcte). La chaîne 'KO' n'ayant pas de propriété `type`, on obtient d'abord `undefined`. Et l'événement est passé en deuxième position, là où `report` attend `info`.

Le problème, c'est que `bind` ne s'attend pas à recevoir un objet événement qui doive être passé « en préfixe » : il préfixe avec les arguments qu'on lui a fournis, et ajoute les arguments d'exécution *ensuite*.

En revanche, `bindAsEventListener` sait parfaitement quel est son rôle, et collera l'événement reçu à l'exécution en première position, conformément aux attentes du gestionnaire. Il passe bien l'événement de clic en premier (sa propriété `type` indiquera le type d'événement comme chaîne de caractères, ici '`click`'), et l'argument pré-rempli, '`OK`', en second.

C'est le *seul* cas où l'on doit passer par `bindAsEventListener` : on pré-remplit des arguments pour un gestionnaire d'événements, qui attend donc l'événement en premier. Si on ne pré-remplit rien, ou qu'il ne s'agit pas d'un gestionnaire exigeant un argument dynamique en première position, `bind` suffit.

Plus de puissance pour les fonctions

Prototype 1.6 a doté les fonctions (et donc les méthodes) d'une série de méthodes puissantes pour manipuler les arguments, « décorer » une méthode existante (au sens du motif de conception Décorateur et de la programmation orientée aspect), et différer l'exécution.

SYNTAXE

```
fonction.argumentNames() -> tableauDeChaines
fonction.curry(arg[, arg...]) -> nouvelleFonction
fonction.delay(secondes[, arg...]) -> timer
fonction.defer([arg...]) -> timer
fonction.wrap(decorateur) -> nouvelleFonction
fonction.methodize() -> versionMéthodisée
```

Que de nouveautés ! Il s'agit souvent de fonctionnalités dont Prototype Core a eu besoin de façon récurrente en faisant évoluer Prototype, et qui ont été formalisées et rendues publiques.

On peut demander à une fonction la liste de ses paramètres avec la méthode `argumentNames()`, qui renvoie un tableau des noms des arguments basé sur l'examen du code source de cette méthode, et donc sur sa déclaration. Attention : le bon fonctionnement n'est pas garanti pour les méthodes natives de JavaScript, qui fournissent rarement une représentation textuelle suffisante.

```
Enumerable.eachSlice.argumentNames()
// => ['number', 'iterator', 'context']
$A.argumentNames()
// => ['iterable']
(5).succ.argumentNames()
// => [] -- il n'y en a aucun
[].indexOf.argumentNames()
// => [] -- les méthodes natives gardent leurs secrets...
```

La méthode `curry()` permet de réaliser une application partielle, c'est-à-dire de pré-remplir un ou plusieurs des premiers arguments d'une méthode, sans pour autant recourir à `bind()` et donc modifier le contexte d'exécution ; il est préférable d'utiliser `curry()` quand on ne souhaite pas modifier la sémantique de `this` par ailleurs.

```
String.prototype.triple = String.prototype.times.curry(3);
'hey! '.triple()
// => 'hey! hey! hey!'
```

Pour différer l'exécution d'une fonction, Prototype fournit deux méthodes : `delay` et `defer`. La première prend un nombre de secondes en premier argument, suivi des éventuels arguments à passer à la fonction le moment venu. Le délai peut bien entendu être inférieur à 1 (par exemple, 0.2 pour 200 millisecondes).

La méthode `defer` représente un cas courant : le *déferrement minimal*, en étant équivalente à un appel à `delay` pré-rempli pour 0,01 seconde. À quoi sert-elle ? Vous savez peut-être que pour le moment, JavaScript est systématiquement « mono-thread », au sens où chaque document ne peut exécuter qu'une instruction JavaScript à un instant t : il n'y a pas de threads multiples, pas d'exécution « parallèle ».

De ce fait, il arrive fréquemment qu'un script réalise des opérations qui vont avoir un impact sur le DOM (par exemple une insertion de fragment HTML), mais que la suite du script ait besoin que le navigateur ait traité l'impact en question avant de continuer (c'est-à-dire que les nouveaux éléments existent bien dans le DOM) ; pour que le navigateur ait ce temps, l'interprétation JavaScript doit généralement terminer la fonction courante. C'est pourquoi le traitement « postérieur à l'impact » peut alors être collé dans une fonction anonyme qui sera différée.

Le code ci-dessous pourrait en effet ne pas fonctionner :

```
function insertAndHighlight(newId, newItem) {
  $('container').insert(newItem);
  // À ce stade, l'élément dans le fragment newItem dont l'ID
  // est passé dans newId n'existe peut-être pas encore dans le
  // DOM : le navigateur n'a pas "eu le temps de respirer"...
```

```
    $(newId).highlight();  
}
```

En revanche, la version suivante est « garantie » :

```
function insertAndHighlight(newId, newItem) {  
    $('container').insert(newItem);  
    (function() {  
        $(newId).highlight();  
    }).defer();  
}
```

Dans la mesure où ici on diffère une seule méthode et pas un bloc de code complexe, on peut même appeler `defer` directement sur la fonction (attention, si c'est une méthode qui a besoin de `this`, il faudra *binder* d'abord ! Ici on utilisera la forme « fonction » pour éviter cela).

```
function insertAndHighlight(newId, newItem) {  
    $('container').insert(newItem);  
    Element.highlight.defer(newId);  
}
```

Il est utile de noter que les méthodes `defer` et `delay` renvoient le *timer* classique (issu d'un appel à `setTimeout`) utilisé pour différer l'appel, et qu'on peut donc technique-ment « annuler un appel différé » avant son exécution en appelant `clearTimeout` sur le *timer* ainsi renvoyé. Mais lorsqu'on veut ce type de contrôle, on utilise générale-ment explicitement `setTimeout` dès le début...

La méthode `wrap` permet de « décorer » une fonction en en produisant une autre qui, elle, récupérera d'une part les arguments et d'autre part une référence vers la méthode d'origine. On peut ainsi, par exemple, filtrer ou convertir certains arguments, cer-taines valeurs de retour, et d'une manière générale implémenter tout comportement complémentaire avant ou après le code d'origine. Il s'agit là d'une des facettes de la programmation orientée aspect (AOP, *Aspect-Oriented Programming*).

Le principe est simple : on appelle `wrap` sur la fonction d'origine en lui passant la fonction qui va l'enrober ; cette dernière recevra dans son premier argument, généralement appelé `proceed` par convention, une référence sur la fonction d'origine. Les arguments suivants seront ceux passés à l'exécution par le code appelant.

Imaginons par exemple qu'on veuille changer le comportement classique de la méthode `capitalize` ajoutée par Prototype aux chaînes de caractères. Par défaut, cette méthode place la première lettre en majuscule et le reste en minuscules :

```
'dommage Éliane'.capitalize() // => 'Dommage éliane'
```

Nous préférerions qu'on puisse aussi capitaliser chaque mot, en passant par exemple un argument optionnel `true` à `capitalize`. Pour cela, nous allons devoir enrober l'implémentation d'origine avec notre comportement spécifique :

```
String.prototype.capitalize = String.prototype.capitalize.wrap(
  function(proceed, eachWord) {
    if (eachWord && this.include(' ')) {
      // capitalize sur chaque mot du texte
      return this.split(' ').invoke('capitalize').join(' ');
    } else {
      // on appelle la fonction d'origine
      return proceed();
    }
  });
'hello world'.capitalize()      // => 'Hello world'
'hello world'.capitalize(true) // => 'Hello World'
```

C'est un peu avancé comme système, mais peut se révéler très puissant lorsqu'on met en place des systèmes de plug-ins qui, par définition, ont besoin d'adapter ou d'améliorer le comportement classique d'une base de code existante.

Pour finir, un mot sur `methodize`. Prototype a un motif de code fréquent qui consiste à proposer des fonctionnalités sous la forme de fonctions normales acceptant leur objet comme premier argument, et à les proposer aussi comme « méthodes » de ces éléments. Il en va ainsi pour les méthodes étendues des éléments du DOM, par exemple :

```
Element.hide(monElement)
Element.addClassNames(monElement, 'working');
// ou :
monElement.hide()
monElement.addClassNames('working');
```

La deuxième forme est souvent plus agréable, notamment parce qu'elle « fait plus objet » et permet le *chainage d'appels*, plus compact :

```
Element.addClassNames(Element.hide(monElement), 'working')
// devient :
monElement.hide().addClassNames('working');
```

Lorsqu'on dispose de fonctions avec ce type de signature, on peut fournir ce genre de comportement en les « méthodisant » : la fonction possède alors une version dérivée, unique (les appels suivants à `methodize` renverront la version dérivée réalisée la première fois, au lieu d'une nouvelle fonction à chaque appel), qui passera en fait `this` comme premier argument à la fonction d'origine.

```
// Fonction définie de façon classique pour une raison quelconque...
function processCoolObj(coolObj, prefix, count) {
    // ...
}
// On en fait une méthode !
CoolObj.prototype.process = processCoolObj.methodize();
unObjetCool.process('préfixe', 5);
```

Attention en revanche au contexte d'utilisation ; pour les méthodes étendues des éléments DOM par exemple, Prototype a un mécanisme global prédefini encore plus puissant, comme nous le verrons plus loin.

De drôles de numéros

Le vénérable objet `Number`, utilisé par tous les nombres, y compris les littéraux, dispose de quelques menues extensions.

SYNTAXE

```
variableNombre.abs() -> ValeurAbsolue
variableNombre.ceil() -> ArrondiHaut
variableNombre.floor() -> ArrondiBas
variableNombre.round() -> ArrondiProche
variableNombre.toColorPart() -> HexaDeuxCaractères
variableNombre.succ() -> nombreSuivant
variableNombre.times(function(index) { ... })
variableNombre.toPaddedString(longueur[, base = 10]) -> Chaîne
(littoralEntier).toColorPart()
(littoralEntier)...
```

Pour commencer, quatre méthodes normalement appelées explicitement sur l'objet natif `Math` sont ajoutées aux nombres pour plus d'orientation objet : ce sont `abs`, `ceil`, `floor` et `round`.

Tout d'abord la méthode `toColorPart()`, qui renvoie la représentation hexadécimale du nombre. Elle est surtout faite pour être utilisée sur des nombres allant de 0 à 255, et destinée à composer la représentation CSS d'une couleur. Voici un exemple typique avec `Array.inject`, que nous étudierons plus loin :

```
var rgb = [ 128, 255, 0 ];
var cssColor = rgb.inject('#', function(s, comp) {
    return s + comp.toColorPart();
});
// cssColor == '#80ff00'
```

Les méthodes `succ()` et `times(iterator)` permettent aux nombres de réaliser des itérations dans le plus pur style Ruby. En effet, la méthode `times` synthétise une itération de 0 jusqu'au nombre `times` (exclu, ou borne ouverte si vous avez l'esprit matheux). Elle utilise pour ce faire un `ObjectRange`, qui a besoin que l'objet d'origine implémente une méthode `succ`, laquelle fournit la valeur suivante à chaque tour (ici, `succ()` renvoie la valeur appelante plus un).

Ainsi, pour réaliser une boucle de zéro à `n` exclu, au lieu de faire :

```
for (var index = 0; index < n; ++index)
    // code
```

On peut faire :

```
n.times(function(index)) {
    // code
}
```

Ce qui est plutôt lisible, vous ne trouvez pas ? « *n times* »...

Enfin, il est possible d'obtenir rapidement un nombre sous la forme d'un texte d'une taille minimale donnée, rempli à gauche par des zéros, et dans la base de votre choix (entre 2 et 36, avec 10 par défaut), à l'aide de `toPaddedString()`.

```
(42).toPaddedString(4)      // => '0042'
(42).toPaddedString(3, 16)  // => '02a'
(42).toPaddedString(4, 2)   // => '101010'
```

Attention toutefois : en raison de la syntaxe des nombres, qui utilisent déjà le point (.) comme séparateur décimal, on ne peut pas invoquer une méthode directement sur un littéral entier : `5.succ()` génère une erreur de syntaxe. On doit donc, comme mentionné plus haut dans le bloc de syntaxe, recourir à une astuce pour lever l'ambiguïté au niveau de l'analyseur. La plus simple est de protéger le nombre par des parenthèses. Il en existe deux autres, que je vous laisse chercher si vous êtes du genre curieux...

Prototype en a dans le String

`String` est l'un des objets les plus enrichis par Prototype. L'autre est `Array`, que nous examinerons tout à l'heure. Il faut avouer que malgré la relative richesse de ses méthodes originelles, `String` présente pas mal de lacunes au regard de son usage courant. En raison du grand nombre d'extensions, j'ai découpé l'examen des nouvelles méthodes par thème.

Toutes les méthodes de modification renvoient en réalité une nouvelle version modifiée : elles ne touchent pas à la chaîne de caractères originale.

Suppression de caractères : `strip`, `stripTags`, `stripScripts`, `truncate`

Ces quatre petites méthodes renvoient une version purgée ou tronquée de la chaîne fournie.

SYNTAXE

```
chaine.strip() -> chaine
chaine.stripTags() -> chaine
chaine.stripScripts() -> chaine
chaine.truncate([longueur = 30[, troncature = '...']]) -> chaine
```

La méthode `strip()` équivaut au plus classique `trim()` : elle retire les espacements (*whitespace*) en début et en fin de texte.

La méthode `stripTags()` purge le texte de toute balise ouvrante ou fermante, attributs compris. Ne pas confondre avec `escapeHTML`.

La méthode `stripScripts()` se contente de supprimer les balises `script` et leur contenu, ce qui est très utile en termes de sécurité. Prototype s'en sert aussi beaucoup en interne, pour insérer du contenu HTML : il retire systématiquement les éléments `<script>` pour les évaluer à part, immédiatement après insertion.

La méthode `truncate` est un tout petit peu plus complexe, car elle dispose de deux arguments optionnels. Il s'agit ici de tronquer un texte à une longueur maximale, en remplaçant la partie tronquée par un texte de substitution. On imagine facilement l'intérêt de cette méthode dans des affichages à largeur fixe. L'argument `longueur`, qui vaut 30 par défaut, donne la longueur maximale du texte résultat (texte de troncature compris), et le second argument indique le texte de troncature, qui vaut par défaut '...' (trois caractères point).

Voici quelques exemples d'utilisation :

Listing 4.6 Méthodes de retrait de caractères

```
var spacedOutText = ' Bonjour monde\n';
var markup = '<h1>Texte balisé</h1>\n' +
    '<p>Vous voyez, il y a des <strong>balises</strong>.</p>';
var scriptMarkup = '<h1>Texte balisé</h1>\n' +
    '<p>Vous voyez, il y a des <strong>balises</strong>.</p>\n' +
    '<script type="text/javascript">\n' +
    'window.location.href="http://site-de-pirate.com/pique_mon_ip";\n' +
    '</script>\n' +
    '<p>Fin du balisage</p>';
var longText = 'Ceci est un texte un peu trop long pour moi';
```

```

spacedOutText.strip()
// 'Bonjour monde'
markup.stripTags()
// 'Texte balisé\nVous voyez, il y a des balises.'
scriptMarkup.stripTags()
// 'Texte balisé\nVous voyez, il y a des balises.\n\n' +
// 'window.location.href = ...\\n\\n' +
// 'Fin du balisage'
scriptMarkup.stripScripts()
// '<h1>Texte balisé</h1>\\n' +
// '<p>Vous voyez, il y a des <strong>balises</strong>.</p>\\n' +
// '\\n<p>Fin du balisage</p>'
longText.truncate()
// => 'Ceci est un texte un peu tr...'
longText.truncate(42)
// => 'Ceci est un texte un peu trop long pour...'
longText.truncate(42, '...')
// => 'Ceci est un texte un peu trop long pour m...'

```

Transformations : `camelize`, `capitalize`, `dasherize`, `escapeHTML`, `gsub`, `interpret`, `sub`, `underscore`, `unescapeHTML`

Ces méthodes permettent d'effectuer des remplacements à comportement variable ou fixe sur le texte.

SYNTAXE

```

chaine.camelize() -> chaine
chaine.capitalize() -> chaine
chaine.dasherize() -> chaine
chaine.escapeHTML() -> chaine
chaine.gsub(pattern, replacement|iterator) -> chaine
String.interpret(chaine) -> chaine
chaine.sub(pattern, replacement|iterator[, count = 1]) -> chaine
chaine.underscore() -> chaine
chaine.unescapeHTML() -> chaine

```

Commençons par les plus simples. La méthode `escapeHTML()` « désamorce » un code HTML : le HTML est utilisé littéralement au lieu d'être interprété par le navigateur. La méthode `unescapeHTML()` fait exactement l'inverse.

Prototype fournit par ailleurs une série de méthodes de conversion entre diverses formes de découpage de texte. La méthode `camelize()` transforme un texte minuscule composé de parties séparées par des tirets (-) en texte *CamelCase* minuscule. Il ne s'agit pas ici d'une transformation destinée aux textes à destination des utilisateurs : on cible spécifiquement le passage d'un nom de propriété CSS à son équivalent dans

le DOM niveau 2 Style. D'ailleurs, cette méthode est utilisée en interne par les méthodes `getStyle` et `setStyle` de `Element.Methods`, que nous étudierons plus loin dans ce chapitre.

La méthode `capitalize` met le premier caractère en majuscule et le reste en minuscules. Quant à `dasherize`, elle remplace simplement les *underscores* (`_`) par des tirets. La méthode `underscore` est surtout utilisée pour passer d'une chaîne « camelisée » à une chaîne découpée par des *underscores*. Remarquez qu'en combinant `underscore` et `dasherize`, on a donc l'inverse de `camelize`...

La méthode « statique » `String.interpret` sert juste de garde-fou contre `null` et `undefined`, en s'assurant d'avoir une chaîne de caractères au bout du compte : la chaîne vide pour `null` et `undefined`, et le résultat d'une conversion `String` telle que prévue par JavaScript pour les autres cas.

Les méthodes les plus avancées sont `sub` et `gsub`. En apparence, elles semblent globalement équivalentes à la méthode existante `replace`. Elles sont en réalité un peu plus puissantes.

Prenons d'abord `gsub`. Elle va remplacer toutes les occurrences du motif `pattern` trouvées dans la chaîne. Ce motif est une expression rationnelle fournie de préférence directement comme objet `RegExp`, par exemple `/w+/`. Ce qui peut varier, c'est le remplacement. On peut passer **un texte ou un itérateur**. Dans le texte, il est possible de référencer les groupes isolés par le motif à l'aide d'une syntaxe inspirée de Ruby : `#{{numéro}}`. L'itérateur, en revanche, sera appelé pour chaque correspondance, avec l'objet représentant la correspondance en argument. Il doit renvoyer le texte de remplacement.

Beaucoup de gens ignorent que la fonction native `String.match` renvoie non pas une chaîne de caractères, mais un tableau qui décrit la correspondance, dont l'index 0 renvoie la correspondance totale, et dont les indices supplémentaires renvoient les groupes isolés par les couples de parenthèses du motif.

Voici un exemple :

```
var md = 'salut les grenouilles'.match(/^(.)(\w+)/);
md[0] // => 'salut'
md[1] // => 's'
md[2] // => 'alut'
```

Enfin, un mot sur la méthode `sub` : elle est similaire à `gsub`, à ceci près qu'elle peut ne remplacer qu'une partie des occurrences (par défaut, seulement la première), à l'aide de son argument optionnel `count`.

Voyons quelques exemples de nos nouvelles méthodes.

Listing 4.7 Transformations de texte

```
'border'.camelize()
// => 'border'
'border-style'.camelize()
// => 'borderStyle'
'border-left-color'.camelize()
// => 'borderLeftColor'
'border_style_color'.dasherize()
// => 'border-style-color'
'borderStyleColor'.underscore()
// => 'border_style_color'
'borderStyleColor'.underscore().dasherize()
// => 'border-style-color'
'prototype Core'.capitalize()
// => 'Prototype core'
'<h1>Un joli titre</h1>'.escapeHTML()
// => '&lt;h1&gt;Un joli titre&lt;/h1&gt;'
'&lt;h1&gt;Un joli titre&lt;/h1&gt;'.unescapeHTML()
// => '<h1>Un joli titre</h1>'
'je m\'appelle charles-edouard'.gsub(/aeiouy/, '-')
// => 'j- m\'-pp-ll- ch-rl-s--d---rd'
'je m\'appelle charles-edouard'.gsub(/aeiouy/, '[#{0}]')
// => 'j[e] m\'[a]pp[e]ll[e] ch[a]rl[e]s-[e]d[o][u][a]rd'
'je m\'appelle charles-edouard'.gsub(/aeiouy/, '\##{0}')
// => 'j#e m\'#app#ell#e ch#arl#es-#ed#o#u#ard'
'charles-edouard de la prutenatilde'.gsub(/\w+/,
    function(match) {
        return match[0].charAt(0).toUpperCase() +
            match[0].substring(1).toLowerCase()
    }
// => 'Charles-Edouard De La Prutenatilde'
'charles-edouard de la prutenatilde'.sub(/\w+/, '[#{0}]')
// => '[charles]-edouard de la prutenatilde'
'charles-edouard de la prutenatilde'.sub(/\w+/, '[#{0}]', 2)
// => '[charles]-[edouard] de la prutenatilde'
```

Fragments de scripts : extractScripts, evalScripts

La méthode `extractScripts` est presque exactement l'inverse de `stripScripts` : elle renvoie un tableau des portions de script du texte (sans leurs balises `script`). La méthode `evalScripts` pousse cette logique un cran plus loin : elle extrait les scripts et les exécute à tour de rôle avec `eval`.

SYNTAXE

```
chaine.extractScripts() -> [texteScript...]
chaine.evalScripts() -> [résultatScript...]
```

Il est assez probable que vous n'utiliserez jamais ces méthodes directement. Elles sont en revanche très utilisées, en interne, par les mécanismes d'insertion (espace de noms `Insertion`, décrit vers la fin du chapitre) et sur les contenus (X)HTML récupérés par les requêtes Ajax.

Conversions et extractions : `scan`, `toQueryParams`, `parseQuery`, `toArray`, `inspect`, `succ`

Commençons par les méthodes simples. Nous avons déjà évoqué `toArray()` en étudiant la fonction `$A`. Une `String` peut ainsi être traitée comme un tableau de caractères.

SYNTAXE

```
chaine.inspect([useDoubleQuotes = false]) -> chaine
chaine.interpolate(objet) -> chaine
chaine.parseQuery/toQueryParams() -> objet
chaine.scan(pattern, iterator) -> LaMemeChaine
chaine.succ() -> chaine
chaine.times(nombre) -> chaine
chaine.toArray() -> [caractère...]
```

La méthode `inspect()` a aussi été vue lorsque j'ai présenté `Object.inspect`. Ici, elle « échappe » simplement les *backslashes* et apostrophes, encadrant le tout entre guillemets simples ou doubles (je trouve d'ailleurs que ce n'est pas suffisant : à mon goût, elle devrait aussi échapper les retours chariot, sauts de ligne et tabulations).

La méthode `parseQuery` analyse une *query string*, cette portion d'une URL qui démarre avec le point d'interrogation (?) et s'arrête éventuellement avec l'ancre (#). C'est là que sont situés les paramètres transmis en GET. La méthode `parseQuery` extrait cette portion d'une URL et la découpe pour renvoyer un objet avec une propriété par paramètre, et bien sûr les valeurs correctement définies pour chaque propriété.

La méthode `toQueryParams` est en réalité un *alias* de `parseQuery` : les deux noms référencent la même méthode. Il s'agit là de l'opération réciproque de `Object.toQueryString`, méthode de conversion qui part d'un objet pour produire sa « sérialisation » sous forme d'une *query string*, fonction que Prototype utilise par exemple en interne pour permettre à ses requêteurs Ajax d'accepter des objets structurés comme description des paramètres de la requête.

Apparue en 1.5.1, `interpolate` permet d'éviter la création explicite d'un objet `Template` à usage unique pour utiliser une chaîne de motifs sur un objet. Nous verrons la classe `Template` plus loin dans ce chapitre, mais la prochaine série d'exemples vous donnera déjà une idée...

Quant à la méthode `times`, c'est une petite méthode de confort servant à « multiplier » une chaîne, comme on peut le faire en Ruby. C'est très pratique pour produire des lignes de séparation, par exemple : `'-'.times(80)` et le tour est joué.

Enfin, la méthode `scan` est une variante restreinte de `gsub`, qui ne sert qu'à transmettre chaque correspondance d'un motif à un itérateur. Elle renvoie la chaîne d'origine au lieu d'un tableau de résultats.

Des p'tits exemples ? Allez, d'accord.

Listing 4.8 Conversions et extractions

```
var query = '?firstName=Christophe&age=28'.parseQuery();
alert(Object.inspect($H(query)));
// => '#<Hash:{'firstName': 'Christophe', 'age': 28}>'
alert(Object.inspect('salut'.toArray()));
// => ['s', 'a', 'l', 'u', 't']
alert(Object.inspect("Salut les p'tits loups \\\o/ !"));
// => 'Salut les p\'tits loups \\\o/ !'
var oCounts = [];
'foo boo boz'.scan(/\o+/, function(match) {
  oCounts.push(match[0].length);
});
alert(Object.inspect(oCounts));
// => [2, 2, 1]
```

Un mot sur la méthode `succ`, qui a un statut un peu particulier : elle ne sert qu'à rendre les chaînes de caractères énumérables au travers d'intervalles représentés par des objets `ObjectRange`. Mais elle se contente de remplacer le dernier caractère de la chaîne par le caractère qui lui succède dans la table Unicode. Ce n'est donc pas très utile en tant que tel.

Vous trouverez une version alternative qui gère correctement le « bouclage » (par exemple, `'wiz'.succ()` donnera bien `'wja'`, et non `'wi{...}'`) dans un billet du blog pour mon livre dédié à Prototype et script.aculo.us : <http://www.thebungeebook.net/2008/04/13/neuron-workout-solutions-5/>.

Inspection : `blank`, `empty`, `endsWith`, `include`, `startsWith`

SYNTAXE

```
chaine.blank() -> booléen
empty() -> booléen
endsWith(chaine) -> booléen
include(chaine) -> booléen
startsWith(chaine) -> booléen
```

Prototype fournit quelques méthodes de confort pour l'examen de chaînes de caractères : `empty` indique que la chaîne est vide, `blank` qu'elle est vide ou ne contient que des espacements (espaces, retours chariot, tabulations...). `startsWith`, `endsWith` et `include` permettent de déterminer que la chaîne sur laquelle on appelle la méthode démarre avec, se termine par ou contient la chaîne passée en argument, respectivement.

Des tableaux surpuissants !

Avant toute chose, il faut dire que `Array` incorpore le module `Enumerable`, que nous allons voir dans la section dédiée aux nouveautés apportées par Prototype. D'entrée de jeu, un tableau dispose donc de toutes les possibilités fournies par ce module, ce qui fait déjà beaucoup.

Cette section se penche sur les méthodes qui sont ajoutées *en plus* de celles du module `Enumerable`. Et il y en a beaucoup, que j'ai classées par thème.

Conversions : from, inspect

La méthode `Array.from` est un alias de la fonction `$A`. Vous pouvez utiliser alternativement l'une ou l'autre syntaxe, suivant votre esthétique personnelle.

La méthode `inspect()` a déjà été vue en étudiant `Object.inspect`. Elle renvoie ici une représentation entre crochets, les éléments étant séparés par une virgule et un espace, chaque élément étant représenté par un `Object.inspect` sur sa valeur. On obtient un texte très proche de l'équivalent littéral JavaScript.

SYNTAXE

```
$A/Array.from(obj) -> tableau
tableau.inspect() -> chaine
```

Quelques exemples :

```
$A('hello')
// => ['h', 'e', 'l', 'l', 'o']
$A({0: 'c\'est', 1: 'vraiment', 2: 'bizarre', length: 3})
// => ['c\'est', 'vraiment', 'bizarre']
[ 42, "hello", Number.POSITIVE_INFINITY, [] ].inspect()
// => [42, 'hello', Infinity, []]
```

Extractions : first, last, indexOf, lastIndexOf

Quelques petites méthodes toutes simples ont été ajoutées pour unifier l'accès aux éléments.

SYNTAXE

```
tableau.first() -> objet
tableau.last() -> objet
tableau.indexOf(obj[, start = 0]) -> nombre
tableau.lastIndexOf(obj[, start = length -1])
```

La méthode `first()` renvoie le premier élément, ou `undefined` si le tableau est vide. La méthode `last()` raccourt enfin la syntaxe pénible `tab[tab.length - 1]`, pour renvoyer le dernier élément (ou `undefined` si le tableau est vide).

La méthode `indexOf`, qui n'est définie par Prototype que si l'implémentation JavaScript du navigateur ne prend pas en charge la méthode native homonyme, raccourt l'éternelle boucle de recherche en renvoyant la position de l'argument dans le tableau. La comparaison se fait avec l'opérateur `==`, et la fonction renvoie `-1` si la valeur n'a pas été trouvée. On peut choisir de démarrer la recherche plus loin que le début, en précisant une position comme deuxième argument.

La méthode `lastIndexOf`, définie dans les mêmes conditions qu'`indexOf`, effectue la recherche dans le sens inverse, et peut également ignorer un segment final de la chaîne plutôt que de partir « de la fin », à l'aide d'un deuxième argument positionnel.

```
[] .first()
// => undefined
[1, 2, 3] .first()
// => 1
[1, 2, 3] .last()
// => 3
$w('salut les grenouilles') .indexOf('les')
// => 1
$w('salut les grenouilles') .indexOf('LES')
// => -1
$w('salut les grenouilles') .indexOf('les', 2);
// => -1
$w('salut les gars les filles') .lastIndexOf('les')
// => 3
$w('salut les gars les filles') .lastIndexOf('les', 3)
// => 1
```

Transformations : `clear`, `compact`, `flatten`, `without`, `reverse`, `reduce`, `uniq`, `intersect`

Pour finir, on dispose de plusieurs méthodes transformant le contenu du tableau. Certaines produisent un nouveau tableau (résultat `tableau` dans la syntaxe), d'autres modifient le tableau d'origine.

SYNTAXE

```
tableau.clear() -> LeMemeTableauMaisVide
tableau.compact() -> tableau
tableau.flatten() -> tableau
tableau.intersect(tableau2) -> tableau
tableau.reduce() -> tableau | valeur
tableau.reverse([inline = true]) -> LeMemeOuUnAutre
tableau.uniq([preSorted = false]) -> tableau
tableau.without(obj...) -> tableau
```

La méthode `clear()` vide le tableau en ramenant sa taille à zéro, et renvoie le tableau lui-même.

La méthode `compact()` crée un nouveau tableau équivalent à l'original dont on aurait retiré tous les éléments `null` et `undefined`.

Très sympathique, la méthode `flatten()`, abondamment utilisée en interne par Prototype : elle « aplatis » un tableau de tableaux, quel qu'en soit le niveau de profondeur. On obtient l'équivalent d'un parcours en profondeur du tableau d'origine. Nous allons l'illustrer dans les exemples à suivre.

La méthode `without` est un filtre bien pratique, qui produit un nouveau tableau équivalent à l'original purgé des arguments passés.

La méthode `reverse` a deux comportements, suivant qu'on lui passe expressément `false` ou non. Dans le premier cas elle produit un nouveau tableau qui est l'inversion (ordre inverse) de l'original. Dans tous les autres, elle inverse directement l'original et le renvoie.

La méthode `reduce` réduit un tableau n'ayant qu'un élément à cet élément lui-même, et un tableau vide à `undefined`.

La méthode `uniq` renvoie le tableau débarrassé de ses doublons. Le tableau n'a pas besoin d'être trié et son ordre n'est pas altéré, mais s'il est *déjà* trié, vous gagnerez en performance à l'indiquer à `uniq` en lui passant `true` en argument (surtout si le tableau est grand).

Enfin, la méthode `intersect` renvoie l'intersection de deux tableaux, c'est-à-dire les éléments du premier qui appartiennent aussi au second (sur la base de l'opérateur de comparaison stricte `==`), sans doublons et dans l'ordre du premier tableau (celui sur lequel on a appelé `intersect`).

Voici quelques exemples de ces méthodes de transformation :

Listing 4.9 Transformations de tableaux

```
var simple = [ 42, 'salut', NaN, 'les', null, 'grenouilles' ];
var complexe = [ 42, [ 'salut', [ NaN ], 'les' ], null,
    [[[ 'grenouilles' ]]] ];

simple.reverse();
// => simple : [ 'grenouilles', null, 'les', NaN, 'salut', 42 ]
simple.reverse(false)
// => [ 42, 'salut', NaN, 'les', null, 'grenouilles' ]
// => simple : [ 'grenouilles', null, 'les', NaN, 'salut', 42 ]

var simple2 = complexe.flatten();
// => simple2 : [ 42, 'salut', NaN, 'les', null, 'grenouilles' ]

simple2 = simple2.without(NaN, 'les').compact();
// => simple2 : [ 42, 'salut', NaN, 'grenouilles' ]

simple2.clear();
// => simple2 : []

[].reduce()      // => undefined
[1].reduce()    // => 1
[1, 2].reduce() // => [1, 2]

[1, 2, 3, 7, 2, 5, 7, 4, 8].uniq() // => [1, 2, 3, 7, 5, 4, 8]
[1, 2, 2, 3, 4, 5, 7, 7, 8].uniq(true) // => same, faster

[1, 2, 3, 4, 4].intersect([8, 6, 4, 2]) // => [2, 4]
```

Notez que le `without` n'a pas purgé le `NaN`, ce qui nous rappelle que par définition, `NaN` n'est égal à aucun `Number`, y compris lui-même ! Or, `without` utilise l'opérateur `==`.

Pour terminer, quelques remarques complémentaires sur `Array`. Nous verrons dans un instant le module `Enumerable`, qui définit notamment des versions génériques, basées sur des boucles de parcours, des méthodes `size` et `toArray`. Ces versions sont inutilement lourdes lorsqu'on utilise un `Array`, aussi ce dernier les redéfinit-il de façon efficace dans son contexte (`toArray` renvoie le tableau lui-même, `size` renvoie simplement la propriété `length`). `Array` en profite pour *aliaser* la méthode `clone` à `toArray`, ce qui est « évident » dans son cas précis.

Enfin, sachez qu'Opera a traditionnellement un problème avec la méthode native `concat` des tableaux ; Prototype la redéclare pour ce navigateur afin de pallier le problème.

Modules et objets génériques

À présent que nous avons vu les extensions que Prototype apporte aux objets natifs de JavaScript, il est temps d'explorer les objets, modules et espaces de noms existant hors des objets natifs, en commençant par ceux répondant à des besoins tellement courants qu'ils constituent une sorte de « bibliothèque standard » générique.

Nous commencerons par l'excellent module `Enumerable`, qui fournit de nombreux comportements d'extraction à tout objet fournissant une méthode d'itération spécifique. Nous poursuivrons par `Hash`, `ObjectRange`, `PeriodicalExecuter`, et pour finir, `Template`.

Enumerable, ce héros

`Enumerable` est un module, au sens défini en début de chapitre. Il s'agit d'un ensemble de méthodes injectables dans un type existant (au travers de l'extension de son prototype), qui ne formulent que quelques dépendances sur ce type pour pouvoir fonctionner.

Avec `Enumerable`, la dépendance est simple : il faut que le type, qui est censé être un conteneur de données, dispose d'une méthode `_each` acceptant un itérateur, et que cette méthode passe à l'itérateur en question toutes les données, tour à tour. Sur cette base simple, `Enumerable` construit la pelletee de méthodes décrite ci-dessous.

« Injecter » `Enumerable` à un type satisfaisant cette contrainte est simple avec Prototype :

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

C'est exactement ce qui se passe, en interne, pour `Array`, `ObjectRange` et `Hash`, pour ne citer qu'eux. Ainsi `Array`, par exemple, itère dans l'ordre croissant de ses indices, de zéro à `length - 1`. D'ailleurs, lorsque le navigateur gère la méthode native `forEach` sur `Array`, c'est celle-ci qui est utilisée en interne pour des raisons d'optimisation.

Le module est vaste, aussi ai-je de nouveau classé les méthodes par thème.

L'itération elle-même : `each`

SYNTAXE

```
objetEnumerable.each(itérateur[, contexte])
```

Pour itérer sur un `Enumerable`, on utilise toujours la méthode `each` publique, jamais la méthode `_each` privée, déclarée par le type concret sur lequel on a injecté le module `Enumerable`. En effet, c'est la version publique qui prend en charge l'exception globale `$break` étudiée plus haut dans ce chapitre. C'est donc grâce à elle qu'il est possible de court-circuiter une itération de ce type.

Par ailleurs, `each` passe en réalité **deux arguments** à l'itérateur : la **valeur** elle-même et la **position** dans le conteneur (comme d'habitude, le premier élément est à la position zéro). Par conséquent, toutes les méthodes acceptant un itérateur dans ce module ont le même comportement.

Voici deux exemples simples d'utilisation :

```
['salut', 'les', 'grenouilles'].each(function(s) {
    alert(s);
});
var sum = 0;
$R(1, 20).each(function(n) { sum += n; });
// sum == 210, somme des entiers de 1 à 20 inclus
```

Le contexte d'itération

Depuis la version 1.6, toute méthode itérative d'`Enumerable` accepte en dernier paramètre optionnel un *contexte*, qui joue le rôle de *binding*. Cette évolution a été mise en place pour harmoniser l'API avec les changements actuels de JavaScript, qui prévoit un tel argument pour ses fonctions équivalentes voire homonymes.

Passer ce contexte permet souvent d'éviter d'avoir à recourir à un `bind` qui serait un peu moins efficace, introduisant un niveau de fonction anonyme supplémentaire.

Je ne préciserai pas cet argument final optionnel dans toutes les signatures, ce serait inutilement lourd. Sachez seulement qu'il est là dès l'instant qu'un itérateur figure parmi les arguments.

```
var CoolGuy = {
    name: 'Dude',
    greet: function(whom) {
        alert(this.name + ' dit : Yo ' + whom + '!');
    }
};

// Efficacité correcte
$w('Elodie Diane Clotilde').each(CoolGuy.greet.bind(CoolGuy));

// Efficacité optimale
$w('Elodie Diane Clotilde').each(CoolGuy.greet, CoolGuy);
```

Tests sur le contenu : all, every, any, some, include, member, size

Les méthodes `any` et `all` sont de plus en plus fréquentes dans les bibliothèques de conteneurs ; on les trouve d'ailleurs dans les tableaux natifs pour les navigateurs Mozilla, et il n'est pas exclu que JavaScript 2.0, le prochain standard, les officialise.

SYNTAXE

```
objetEnumerable.all/every([iterator]) -> booléen  
objetEnumerable.any/some([iterator]) -> booléen  
objetEnumerable.include/member([iterator]) -> booléen  
objetEnumerable.size() -> nombre
```

Il s'agit pour la plupart de prédictats permettant de vérifier si tous les éléments d'un conteneur satisfont une condition, ou si au moins un d'entre eux la satisfait. C'est extrêmement pratique dans de nombreux cas, par exemple pour vérifier qu'un tableau ne contient que des nombres, ou que dans une liste censée contenir des éléments du DOM, il y a au moins un élément qui n'est ni `null`, ni `undefined`.

Les deux méthodes acceptent un itérateur optionnel, qu'on fournit généralement. S'il est manquant, on utilise la fonction identité, `Prototype.K`. Les éléments eux-mêmes sont alors traités comme des équivalents booléens (voir le chapitre 2 pour les correspondances).

L'itération est bien sûr court-circuitée dès que son résultat est déterminé (`all` court-circuite au premier élément insatisfaisant, `any` au premier satisfaisant).

Voici l'expression des deux tests cités en exemple un peu plus tôt :

```
if (data.all(function(x) { return 'number' == typeof x; }))  
...  
if (data.any())
```

Notez l'absence d'itérateur au deuxième appel : on teste directement les éléments. Avouez que c'est concis !

Prototype a récemment *aliasé* ces méthodes avec leurs noms officiels dans les versions récentes de JavaScript : `every` et `some`, afin de faciliter la transition ultérieure vers les méthodes natives.

Il existe également une méthode `include` (et son alias `member`, c'est au choix), qui permet de tester qu'un conteneur contient bien un objet particulier. La méthode renvoie un booléen et court-circuite évidemment dès qu'elle a trouvé.

```
$R(1, 20).include(12) // => true  
$A(document.childNodes).include(document.body) // => false
```

Enfin, une méthode `size` donne la taille de l'énumération, mais attention : pour ce faire, elle est contrainte, par défaut, de parcourir toute la séquence, ce qui est souvent sous-optimal. Les types qui incluent `Enumerable` mais disposent d'une alternative performante redéclarent `size`, comme c'est le cas de `Array` qui consulte alors simplement sa propriété native `length`.

Extractions : `detect`, `find`, `filter`, `findAll`, `select`, `grep`, `max`, `min`, `pluck`, `reject`, `partition`

Les objets énumérables disposent de toute une panoplie d'extractions, toutes plus pratiques les unes que les autres.

SYNTAXE

```
objetEnumerable.detect/find(iterateur) -> objet
objetEnumerable.filter/findAll/select(iterateur) -> [objet...]
objetEnumerable.grep(pattern[, iterateur]) -> [objet...]
objetEnumerable.max([iterateur]) -> objet
objetEnumerable.min([iterateur]) -> objet
objetEnumerable.partition([iterateur]) -> [[objV...], [objF...]]
objetEnumerable.pluck(nomPropriete) -> [valeurPropriete...]
objetEnumerable.reject(iterateur) -> [objet...]
```

La méthode `detect` (et son alias `find`) recherche le premier objet satisfaisant l'itérateur dans l'énumération et le renvoie immédiatement (s'il n'a pas été trouvé, elle renvoie `undefined`). Vous voulez la première valeur supérieure à 10 dans un tableau de nombres ?

```
desNombres.find(function(n) { return n > 10; })
```

La méthode `select` (et ses aliases `filter` et `findAll`) ne se contente pas de la première valeur, mais renvoie l'ensemble des valeurs satisfaisant l'itérateur. À ce titre, c'est l'opposée de `reject`, qui renvoie toutes les valeurs *ne satisfaisant pas* l'itérateur.

Si vous avez besoin des deux (la liste des éléments sélectionnés, donc satisfaisant votre prédicat, *et* celle de ceux rejétés, qui ne le satisfont pas), utilisez plutôt `partition` : elle donnera les deux listes en ne parcourant l'énumération qu'une seule fois, ce qui est plus efficace.

Voici comment récupérer tous les textes de plus de 42 caractères, et comment virer les `Nan` :

```
desTextes.select(function(s) { return s.length > 42; })
desNombres.reject(function(n) { return isNaN(n); })
```

Et voici comment découper en une seule itération une série de nombres en pairs et impairs :

```
var numbers = [1, 2, 4, 7, 9, 15, 18, 21, 30, 42];
var result = numbers.partition(function(n) {
    return 0 == n % 2; });
// result[0] => [2, 4, 18, 30, 42]
// result[1] => [1, 7, 9, 15, 21]
```

La méthode `grep` est une variante particulière de `select`. Attention, elle a changé de sémantique depuis la version 1.5 ! Auparavant, elle convertissait les valeurs en `String` et leur appliquait une expression rationnelle. Aujourd'hui, elle applique à chaque valeur la méthode `match` de son argument, laquelle méthode doit renvoyer un booléen, et ne garde que les valeurs ayant produit `true`.

Afin de rester compatible avec l'ancienne version, Prototype *alias* la méthode native `test` de `RegExp` en `match`, ce qui permet bien de toujours passer des expressions rationnelles, mais il faut que les valeurs soient bien des objets `String` pour avoir un traitement fiable...

Pour récupérer les textes ne contenant aucune voyelle, ou les longueurs de ceux contenant au moins trois mots, on procéderait ainsi :

```
desTextes.grep(/^[^aeiouy]*$/i)
desTextes.grep(/(\w+(\b|\W+)){3,}/, function(s) {
    return s.length;
})
```

En revanche, on peut désormais utiliser le mécanisme de `grep` avec tout type de filtre : il suffit que l'objet filtrant fournisse une méthode `match` représentant notre prédicat. C'est notamment le cas des objets `Selector`, qui encapsulent une sélection CSS. On peut donc facilement filtrer une séquence d'éléments DOM pour ne garder que ceux qui correspondent à cette sélection.

```
var filter = new Selector('pre + p');
var postCodePars = paragraphs.grep(filter);
postCodePars.invoke('setStyle', 'background: #ffc');
```

Les méthodes `min` et `max` parlent d'elles-mêmes. Sans itérateur, elles travaillent sur les objets directement, qui doivent donc prendre en charge les opérateurs relationnels `>` et `<`. Un itérateur optionnel permet de rechercher le minimum et le maximum d'une valeur dérivée plutôt que les objets eux-mêmes.

Par exemple, voici comment connaître le premier texte d'une série dans l'ordre lexicographique (celui de la table des caractères), ainsi que la taille du texte le plus long :

```
desTextes.min()
desTextes.max(function(s) { return s.length; })
```

Avec un peu d'astuce, on peut obtenir par exemple le plus grand nombre de voyelles dans les textes :

```
desTextes.max(function(s) {
  return (s.match(/[aeiouy]/ig) || []).length;
});
```

Pour finir, la méthode `pluck` permet de récupérer une propriété particulière pour chaque objet de l'énumération. Cette méthode est très pratique, performante (pas d'appel de méthode à chaque parcours de boucle en interne) et largement utilisée en interne par Prototype. Vous voulez un tableau des tailles de tous les textes ?

```
desTextes.pluck('length')
```

Transformations et calculs : collect, map, inGroupsOf, inject, invoke, sortBy, zip

La frontière entre extraction et transformation est parfois ténue. Si vous estimez qu'une de ces méthodes aurait davantage trouvé sa place à la précédente section, ou réciproquement, vous avez peut-être raison. Tout dépend, en réalité, de l'utilisation concrète, et les possibilités de combinaison ou d'usage... détourné sont potentiellement illimitées.

SYNTAXE

```
objetEnumerable.collect/map(iterateur) -> tableau
objetEnumerable.inGroupsOf(taille[, rempisseur]) -> tableau
objetEnumerable.inject(accumulateur, iterateur) -> valeur
objetEnumerable.invoke(nomMethode[, arg...]) -> tableau
objetEnumerable.sortBy(iterateur) -> tableau
objetEnumerable.zip(equivalentTableau..., iterateur) -> tableau
```

La méthode `map` (et son alias `collect`) crée un nouveau tableau où chaque élément est le résultat de l'application de l'itérateur à son homologue du tableau original. Lorsqu'il s'agit simplement de remplacer un objet par une de ses propriétés, préférez `pluck`, tellement plus simple. Pour un appel homogène de méthode, `invoke` sera le meilleur choix (voir plus loin). Mais pour quelque chose de plus avancé, `map` est incontournable. Vous voulez la parité de tous les nombres d'un tableau ?

```
[1, 2, 5, 8, 14].map(function(n) { return 0 == n % 2; })
// => [false, true, false, true]
```

La méthode `inGroupsOf` découpe une séquence en plusieurs groupes de taille définie ; le dernier n'aura pas forcément assez d'éléments restants, et sera complété en utilisant le deuxième argument, qui vaut par défaut `null`. C'est souvent bien pratique pour créer des colonnes ou matrices d'informations.

```
$R(1,10).inGroupsOf(3)
// => [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, null, null]]
```

Ce comportement repose en réalité sur un *itérateur alternatif* à `each`, qui s'appelle `eachSlice` ; il est plutôt « interne » mais reste garanti dans les prochaines versions, aussi je vous le livre. Au lieu d'itérer élément par élément, `eachSlice` itère groupe d'éléments par groupe d'éléments. Il ne complète pas le dernier groupe en cas de manque.

```
$R(1, 10).eachSlice(3, function(group) {
  alert(group.inspect());
})
// => Affiche "[1, 2, 3]", "[4, 5, 6]", "[7, 8, 9]" puis "[10]"
```

La méthode `inject` est très utile (nous nous en sommes d'ailleurs déjà servi dans des exemples précédents, pour `Number.parseColorPart()` par exemple) : elle permet de construire une valeur à partir de l'ensemble des objets de l'énumération.

On utilise une variable dite *accumulateur*, qui part d'une valeur initiale fournie en premier argument. Ensuite, à chaque valeur de l'énumération, l'itérateur est invoqué avec la valeur actuelle de l'accumulateur comme premier argument, et l'objet courant en second. L'itérateur doit **renvoyer** la nouvelle valeur de l'accumulateur.

La fonction `inject` renvoie la valeur finale de l'accumulateur. Il peut s'agir d'une chaîne de caractères, d'un nombre, d'un booléen... Tout dépend de vos besoins ! Voici par exemple comment établir la somme et le produit internes d'un tableau de nombres :

```
desNombres.inject(0, function(acc, n) { return acc + n; })
desNombres.inject(1, function(acc, n) { return acc * n; })
```

Notez l'importance de la valeur initiale pour l'accumulateur. Je précise que l'itérateur prend en troisième argument la position de l'objet courant dans le tableau, même si on s'en sert plutôt rarement.

La méthode `invoke` n'est pas sans rapport à `pluck`. Là où `pluck` récupère une propriété, `invoke` appelle pour chaque objet de l'énumération la méthode désignée, en lui passant les éventuels arguments supplémentaires, et fournit le tableau des résultats. Si vous souhaitez par exemple obtenir les 10 premiers caractères de chaque texte, vous pourriez utiliser le `map` classique :

```
desTextes.map(function(s) { return s.substring(0, 10); })
```

Ou préférer `invoke`, plus élégant et aussi plus performant :

```
desTextes.invoke('substring', 0, 10);
```

La méthode `sortBy` permet de produire une variante triée du tableau, suivant un critère défini par l'itérateur obligatoire (pour trier les objets directement, on dispose déjà de la méthode native `sort` sur `Array`). Par exemple, on peut vouloir trier des textes selon leur taille :

```
desTextes.sortBy(function(s) { return s.length; })
```

La dernière méthode de cette section, `zip`, est assez complexe. Il s'agit de « coller » des tableaux les uns aux autres, côté à côté si l'on peut dire. La méthode produit dans tous les cas un nouveau tableau résultat.

Imaginons par exemple que vous ayez un tableau d'identifiants utilisateur, et un autre de mots de passe. Vous souhaitez produire un tableau de paires : identifiant + mot de passe. Pour cela, vous avez plusieurs moyens. Si un tableau de tableaux vous convient (tableaux internes à deux éléments), alors c'est tout à fait trivial :

```
var logins = [ 'tdd', 'al', 'nioute', 'doudou' ];
var passwords = [ 'blah', 'koolik', 'linux', 'lachef', 'toto' ];
var auth = logins.zip(passwords);
// => [[ 'tdd', 'blah' ], [ 'al', 'koolik' ],
//       [ 'nioute', 'linux' ], [ 'doudou', 'lachef']]
```

Notez que les éléments superflus des tableaux « zippés » sont ignorés : l'itération prend le tableau appelant comme base.

Peut-être cette structure en tableaux imbriqués vous pose-t-elle problème, et vous préféreriez des objets avec des propriétés nommées. Qu'à cela ne tienne ! Vous pouvez préciser comme dernier argument un itérateur chargé de construire la valeur « zippée » sur la base du tableau interne initialement prévu. Voici notre appel précédent ajusté :

```
var auth = logins.zip(passwords, function(args) {
    return { login: args[0], password: args[1] };
});
alert(auth[0].login + ' / ' + auth[0].password);
// => 'tdd / blah'
```

Notez qu'il est ainsi possible de « zipper » autant de tableaux qu'on le souhaite. Si vous aviez par exemple un tableau supplémentaire de *salt*s pour l'encryptage des mots de passe, vous pourriez tout à fait l'ajouter :

```
var auth = logins.zip(passwords, salts, function(args) {
    return { login: args[0], password: args[1], salt: args[2] };
});
```

Conversions : `toArray`, `entries`, `inspect`

Enfin, il existe quelques conversions simples sur les objets énumérables.

SYNTAXE

```
objetEnumerable.toArray/entries() -> tableau
objetEnumerable.inspect() -> chaine
```

D'abord tout objet énumérable peut être converti en tableau, par exemple avec \$A, puisqu'on dispose d'une méthode `toArray()` (et d'un alias, `entries`). Notez que sur un Array, `toArray()` renvoie donc un clone du tableau d'origine, ce qui peut s'avérer pratique de temps à autre.

Par ailleurs, les objets énumérables fournissent bien sûr une méthode `inspect()`, qui renvoie une représentation textuelle orientée débogage. Pour information, il s'agit en réalité de celle disponible dans Array, encadrée de '#<Enumerable:' et de '>'. Notez que lorsqu'un objet incorporant Enumerable redéfinit sa méthode `inspect()` (comme c'est le cas pour Array et Hash, notamment), la version redéfinie a priorité.

Il suffit parfois d'un peu de Hash

Un Hash est une sorte de tableau associatif entre des clefs et des valeurs. En JavaScript, tout objet peut être considéré comme un *hash*, puisqu'il ne s'agit finalement que d'une série de propriétés définies comme des paires nom + valeur (bon, il y a aussi le prototype, c'est vrai).

L'objet Hash de Prototype fournit cependant un bon nombre de méthodes conçues pour travailler expressément sur une série de paires clef + valeur.

Il faut aussi savoir qu'**un Hash incorpore Enumerable** (et ce n'est pas rien !) L'itérateur reçoit alors comme objet courant une paire nom + valeur sous un format confortable : on peut s'en servir comme un tableau à deux éléments, ou utiliser ses propriétés `key` et `value`.

SYNTAXE

```
$H/Hash.from/new Hash([base]) -> Hash
hash.clone() -> nouveauHash
hash.get(clef) -> valeur
hash.index(valeur) -> clef
hash.inspect() -> chaîne
hash.keys() -> tableau
hash.merge(autreHash) -> nouveauHash
hash.set(clef, valeur) -> mêmeValeur
hash.toObject/toTemplateReplacements() -> objetSimple
hash.toQueryString() -> chaîne
hash.unset(clef) -> ancienneValeurEventuelle
hash.update(autreHash) -> hashModifié
hash.values() -> tableau
```

Hash est typiquement une API qui a beaucoup évolué entre les versions 1.5 et 1.6 de Prototype, pour arriver à son statut plus mûr d'aujourd'hui. D'une part, l'ancienne distinction `$H` / `new Hash` (la deuxième forme n'incorporait pas `Enumerable`) a été supprimée ; d'autre part, en ayant recours à un objet privé comme conteneur, Hash permet désormais d'utiliser absolument n'importe quelle clef ou valeur, sans restriction de type ni de nom, au prix des nouvelles méthodes `get()` et `set()`. Enfin, de nombreuses méthodes complémentaires ont vu le jour.

La méthode `clone()` parle d'elle-même... Les méthodes `keys()` et `values()` renvoient chacune un tableau. Bien entendu, la première renvoie les clefs (en général, les noms des propriétés), tandis que la seconde renvoie les valeurs. La première est généralement beaucoup plus utilisée que la seconde. En effet, quand on a le nom d'une propriété de `obj` dans une variable `key`, la manière la plus directe d'accéder à la valeur est `obj[key]` ! L'ordre du résultat dépend du navigateur, car il est basé sur la boucle native `for...in` dont l'ordre de parcours peut varier.

Par exemple, si vous voulez connaître tous les champs accessibles pour l'objet `navigator` du DOM niveau 0, il suffit de lancer ceci :

```
alert($H(navigator).keys().sort().join('\n'));
```

On ajoute ou modifie une association avec `set()`, qui renvoie la valeur définie pour permettre des utilisations ou affectations à la volée :

```
var result = cache.set('result', someExpensiveComputation());
```

On accède aux valeurs obligatoirement *via* la méthode `get()`, qui renvoie `undefined` si la clef n'est pas trouvée. **Attention !** Ceci constitue un changement d'API non compatible avec les précédentes versions : on ne peut plus écrire ou lire les associations à même l'objet `Hash`, il faut passer par `get()` et `set()`.

Une nouvelle méthode joue d'ailleurs le rôle inverse de `get` : `index()`, qui renvoie la « première » clef associée à une valeur (toujours suivant l'ordre natif du `for...in`), ou `undefined` si la valeur est inconnue au bataillon...

Enfin, pour retirer une association, on utilise `unset`, qui renvoie au passage la valeur de l'association supprimée (ou `undefined` si cette association n'existe pas).

Inutile de s'appesantir sur la méthode `inspect()`, au rôle désormais classique. En revanche, la méthode `merge()` est pratique : elle permet de combiner deux `Hash` pour produire un nouveau `Hash` résultat. Ce dernier contient toutes les clefs des deux `Hash` d'origine. En cas de clef dupliquée, la valeur du `Hash` passé en argument aura écrasé celle du `Hash` d'origine. Si vous avez besoin de modifier le `Hash` lui-même plutôt que d'en générer un nouveau, utilisez plutôt `update()`.

La méthode `toQueryString()` s'avère fort utile lorsque nos scripts doivent composer des paramètres de requête GET, ou un corps URL-encodé de requête POST. En effet, cette méthode construit une représentation textuelle URL-encodée des paires clef + valeur du `Hash`. En construisant sa série de paramètres dans un `Hash` ou un objet anonyme, on peut donc aisément obtenir le texte final (ce qui est plus simple que de le confectionner soi-même). Petit exemple rapide :

```
var params = { title: 'Un Exemple', age: 28 };
var encoded = $H(params).toQueryString();
// => title=Un%20Exemple&age=28
```

Autre option, qui revient strictement au même en interne :

```
var encoded = Object.toQueryString(params);
```

Remarquez qu'il s'agit là de la fonction réciproque de `String.parseQuery`. Nous nous en servirons dans nos prochains chapitres sur Ajax.

Petite remarque au passage : dans Prototype 1.5, on pouvait faire `Hash.toQueryString` (version « statique ») sur des objets quelconques ; désormais, c'est `Object.toQueryString`, comme on vient de le voir.

Enfin, la fonction `toObject()` renvoie un clone du conteneur interne, objet privé dont nous manipulons indirectement les propriétés et leurs valeurs. Le nom n'est pas innocent, comme nous le verrons plus tard : il est notamment utilisé par les nouvelles

fonctionnalités JSON de Prototype, ce qui permet ici de sérialiser un Hash de façon transparente, sur la base de son conteneur privé.

Cette fonction est *aliasée* en `toTemplateReplacements()`, pour des raisons que nous détaillerons dans une prochaine section consacrée à `Template`.

ObjectRange : intervalles d'objets

Il s'agit d'un petit objet simple, qui vise à encapsuler une définition de boucle, généralement numérique. Toutefois, il peut fonctionner sur n'importe quel objet implémentant une opération `succ()` pour passer d'une valeur à la suivante.

SYNTAXE

```
$R(debut, fin[, finExclude = false]) -> intervalle  
intervalle.include(valeur) -> booléen
```

La manière la plus simple de créer un `ObjectRange` est d'utiliser la fonction globale `$R`, déjà vue plus haut dans ce chapitre. Sinon, `new ObjectRange(...)`.

La méthode `include` exige que les objets sous-jacents prennent en charge les opérateurs relationnels `<` et `<=`. Ceci s'entend au sens dynamique de JavaScript, pas au sens statique de compilation des *templates C++*, par exemple : si on n'appelle pas la méthode `include`, le fait que les objets sous-jacents n'implémentent pas un comportement cohérent sur `<` et `<=` est sans importance.

`ObjectRange` incorpore bien entendu `Enumerable`, ce qui fait qu'il permet non seulement de réaliser des itérations, mais aussi de bénéficier de toutes les opérations fournies par le module.

On a déjà vu des exemples d'utilisation à plusieurs reprises dans ce chapitre.

PeriodicalExecuter ne se lasse jamais

SYNTAXE

```
new PeriodicalExecuter(callback, intervalInSeconds)  
pe.stop()
```

S'il vous arrive de devoir exécuter une fonction à intervalles réguliers, `PeriodicalExecuter` est là. Très honnêtement, cet objet n'a qu'une utilité : il évite la réentrance en n'invoquant pas la méthode si son invocation précédente n'est pas terminée.

Pour stopper la répétition, il suffit d'appeler la méthode `stop`. L'objet se passant comme premier argument de la fonction de rappel, celle-ci peut décider d'être la dernière invocation en appelant `stop` sur l'argument.

Attention ! Si vous passez une méthode, pensez à conserver son *binding* correctement. Exemples d'utilisation :

```
new PeriodicalExecuter(function() { window.title += '.'; }, 1);
...
var notifier = $('notifier');
new PeriodicalExecuter(notifier.cleanup.bind(notifier), 5);
```

You devriez ré-évaluer vos modèles

La version 1.5 a vu apparaître un petit objet curieux : `Template`. Cet objet permet de gérer des syntaxes de substitution au sein d'un modèle textuel. Les syntaxes en question ont été considérablement améliorées avec la version 1.6.

SYNTAXE

```
new Template(templateText[, pattern]) -> modèle
modèle.evaluate(scopeObject) -> chaîne
```

Par exemple, si vous examinez la méthode `String.gsub`, vue plus haut dans ce chapitre, vous voyez une syntaxe de type `#{}{propriété}`, avec gestion d'un *backslash* devant le `#` pour désamorcer l'interprétation. Nous avons aussi étudié la méthode `String.interpolate`. Leur travail est réalisé en interne par un objet `Template`. Je vais appeler de tels objets des **modèles**.

Un modèle est défini par deux caractéristiques :

- 1 Son **motif textuel**, qui constitue le patron sur lequel confectionner le texte résultat à chaque évaluation.
- 2 Son **motif de détection**, qui est une expression rationnelle décrivant les portions dynamiques du motif textuel. Si vous souhaitez utiliser le motif par défaut, comme pour `String.gsub` par exemple, vous n'avez pas besoin de passer ce second argument à la construction : le motif par défaut `Template.Pattern` est alors utilisé.

Le principe est cependant toujours le même : le motif de détection vise à isoler un nom de propriété (en l'occurrence un champ, pas une méthode) qui sera récupéré dans l'objet de contexte fourni à l'évaluation. On peut en effet réutiliser un même modèle autant de fois qu'on veut, en appelant sa méthode `evaluate`, à laquelle on fournit l'objet contenant les propriétés à substituer dans le corps du texte.

Voyez l'exemple suivant :

```
var person = { name: 'Élodie', age: 27, height: 165 };
var tpl = new Template('#{name} a #{age} ans et mesure #{height}cm.');
alert(tpl.evaluate(person));
// => 'Élodie a 27 ans et mesure 165cm.'
```

N'est-ce pas sympathique en diable ? On peut même aller plus loin désormais, on bénéficiant des opérateurs . (point) et []. Voici un exemple un peu plus costaud :

```
var people = [
  { name: 'Élodie Jaubert',
    interests: [ 'le patrimoine', 'la culture' ] },
  { name: 'Seth Dillingham' }
];
var tpl = new Template(
  '#{length} personnes. #{0.name} aime #{0.interests[1]}, ' +
  'entre autres choses.');
tpl.evaluate(people)
// => '2 personnes. Élodie aime la culture, entre autres choses.'
// Vous pouvez utiliser [] ou . de façon interchangeable, sauf
// quand le nom de la propriété est vide ou contient un point :
// il faut alors utiliser les crochets.
```

Enfin, `Template` admet qu'il peut arriver des moments où l'on aimerait appeler des méthodes sur l'objet passé à `evaluate`, ou effectuer d'autres comportements alternatifs. Pour permettre à ces objets de faire ce qu'ils souhaitent dans le cadre de `evaluate`, `Template` leur propose de fournir une méthode `toTemplateReplacements`, qui renvoie un objet dont les propriétés ont des valeurs appropriées. Par exemple, un tel objet pourrait avoir des propriétés dont les valeurs seraient les résultats d'appels de méthodes. Voyez plutôt :

```
var student = {
  name: 'Alexis',
  grades: [10, 12, 13.5, 8, 16],

  average: function() {
    return this.grades.inject(0, function(acc, g) { return acc + g }) /
      this.grades.length;
  },
  highest: function() { return this.grades.max(); },
  lowest: function() { return this.grades.min(); },
  toTemplateReplacements: function() {
    var result = Object.clone(this), student = this;
```

```

['average', 'highest', 'lowest'].each(function(methodName) {
  result[methodName] = student[methodName]();
});
return result;
};

var tpl = new Template(
  '#{name} a une moyenne de #{average} (plus bas : #{lowest})';
tpl.evaluate(student);
// => 'Alexis a une moyenne de 11.9 (plus bas : 8)'

```

Voilà notamment pourquoi Hash renvoie un clone de son objet conteneur interne avec `toTemplateReplacements` : cela permet de passer à `evaluate` un Hash au même titre qu'un objet « nu ».

Utiliser un motif de détection personnalisé

Supposons à présent que vous deviez travailler sur les éléments d'un tableau, et que cette syntaxe vous semble trop lourde. Vous pouvez fournir votre propre motif de détection. Seule condition (mais de taille pour les débutants en expressions rationnelles), celui-ci doit isoler trois groupes :

- 1 Le caractère ou l'ancre situé juste avant la portion à substituer.
- 2 L'intégralité de la portion à substituer.
- 3 La partie de la portion à substituer qui constitue le nom de la propriété à récupérer dans l'objet de contexte.

Par exemple, le motif par défaut, `Template.Pattern`, a l'aspect suivant :

```
/(^|.|\\r|\\n)(#\{\.(.*?)\})/
```

On distingue bien les trois groupes :

- 1 `(^|.|\\r|\\n)` peut correspondre au début de texte, à un retour chariot ou saut de ligne, ou un autre caractère quelconque.
- 2 `(#\{\.(.*?)\})` correspond à tout le bloc à substituer, du `#` initial à l'accolade fermante.
- 3 `(.*?)` correspond au nom de la propriété, constitué d'un nombre quelconque de caractères quelconques, à l'exception de l'accolade fermante (le `*` est un *quantificateur réticent*, qui s'arrête le plus tôt possible).

Le premier groupe est nécessaire pour permettre à `evaluate` d'ignorer les portions précédées d'un *backslash*. Le second groupe lui permet justement de laisser inchangé

le texte original dans un tel cas. Le troisième groupe lui donne le nom de la propriété à aller chercher.

Imaginons donc que nous souhaitions une syntaxe à base de dollars : \$index. On considère que les indices sont constitués exclusivement de chiffres arabes. Notez qu'une telle syntaxe, sans caractère délimiteur de fin, empêche de coller de tels chiffres immédiatement à la suite de nos substitutions (ce qui tend à démontrer l'utilité de tels encadrements). Voici notre motif de détection, très inspiré de celui par défaut :

```
/(^|.|\\r|\\n)(\\$((\\d+)))/
```

Voyons le résultat :

```
var data = [ 'Élodie', 27, 1.65 ];
var tpl = new Template('$0 a $1 ans et mesure $2m.',
  /(^|.|\\r|\\n)(\\$((\\d+)))/;
alert(tpl.evaluate(data));
// => 'Élodie a 27 ans et mesure 1.65m.'
```

Notez que si la propriété est introuvable, evaluate lui substitue la chaîne vide ('').

Prise en charge de JSON

Prototype 1.6 a introduit une prise en charge exhaustive de JSON, qu'il s'agisse de sérialiser ou désérialiser des objets natifs (Object, Array, Date, Number, String) ou des classes Prototype (Hash), sans parler de l'interaction avec Ajax.

JSON signifie JavaScript Object Notation. Il s'agit au départ d'une simple restriction des syntaxes de valeurs littérales en JavaScript proposée par Douglas Crockford afin de représenter l'ensemble des types natifs et, par extension, tout type d'objet. JSON ne se préoccupe pas des méthodes : seuls les « champs » sont traités.

JSON joue en quelque sorte le rôle de XML, c'est-à-dire celui d'un format de données structurées. Cependant il se révèle d'une part beaucoup moins verbeux, et d'autre part trivial à exploiter en JavaScript, puisqu'il suffit en effet d'appeler eval sur le texte JSON pour obtenir la grappe d'objets correspondante ! De quoi envoyer XSLT et XPath au placard...

JSON connaît depuis quelque temps un tel engouement qu'on voit apparaître plusieurs mini-standards autour de lui, notamment JSONP, JSONRequest, JSON-RPC et JSON Schema. Ceux-ci sont toutefois bien plus légers que leurs homologues WS-*/SOAP/XML, assurez-vous...

Nous reviendrons sur la pertinence de JSON comme format d'échange, notamment dans le cadre d'échanges Ajax, au chapitre 7. Pour le moment, permettez-moi simplement de faire un rapide tour d'horizon de la prise en charge que fournit Prototype :

SYNTAXE

```
Object.toJSON(objet) -> chaîne
tableau.toJSON() -> chaîne
date.toJSON() -> chaîne
hash.toJSON() -> chaîne
nombre.toJSON() -> chaîne
chaîne.toJSON() -> chaîne
chaîne.isJSON() -> booléen
chaîne.evalJSON(sanitize) -> valeur
```

Les méthodes `toJSON()` renvoient la représentation JSON (donc textuelle) de l'objet sur lequel on les appelle. Les fonctions, la valeur spéciale `undefined` et les types « exotiques » (objets ActiveX sur MSIE par exemple) sont ignorés, et tenter un `Object.toJSON` dessus renverra donc `undefined`.

En-dehors de ceci, les syntaxes sont plutôt évidentes :

- booléens : `'true'`, `'false'`
- `null` : `'null'`
- Date : `'''2008-07-26T18:06:35Z'''` (format W3DTF)
- Chaîne de caractères : `""contenu de la chaîne""`
- Tableau : `'[jsonElement1, jsonElement2, ...jsonElementN]'`
- Nombre : `'lenombre'` ou `'null'` pour un infini, non géré par JSON.
- Objet quelconque : `'{"prop1": jsonValeur1, ..."propN": jsonValeurN}'`
- Hash : comme pour un objet quelconque.

La méthode `isJSON` détermine si une chaîne de caractères contient une sérialisation JSON valide.

Tout ceci traitait de la sérialisation, mais il nous reste maintenant à déserialiser ! C'est le rôle de la méthode `evalJSON()` des chaînes de caractères, qui va évaluer son contenu et renvoyer la structure de données résultat. Par défaut, le contenu est évalué quel que soit son « aspect ». En passant `true` pour le paramètre optionnel `sanitize`, on n'évaluera que si `isJSON()` a donné son `aval`.

Plus de POO classique : classes et héritage

Avec la version 1.6, Prototype a introduit une refonte complète de son système de classes. Jusque là, il fallait se débrouiller avec le faiblard `Class.create` et la polyvalence de `Object.extend...`. On peut désormais facilement réaliser les tâches suivantes :

- définir une classe et ses méthodes d'instance ;
- définir un héritage de classe ;
- incorporer (*mix in*) des modules dans une classe ;
- ajouter *a posteriori* des méthodes d'instance à une classe ;
- ajouter des méthodes statiques à une classe ;
- déterminer le constructeur d'une classe, sa chaîne de classes ancêtres et descendantes ;
- surcharger une méthode et accéder à sa version héritée.

Et la version 1.6.1 devrait fournir des méthodes de confort complémentaires permettant par exemple de lister les méthodes statiques et d'instance, ou de réagir à des événements spéciaux comme l'héritage de classe, l'incorporation de module, etc.

En attendant, voici déjà les composants, simples mais puissants, du système actuel.

SYNTAXE

```
Class.create([parent[, module...],] méthodesInstance) -> Classe
classe.addMethods(méthodesInstance) -> Classe
classe.superclass -> Classe
classe.subclasses -> [Classe...]
objet.constructor -> Classe
Class.Methods
```

On crée une classe, comme en 1.5, avec un appel à `Class.create`. Mais au lieu de fournir simplement les méthodes d'instance, on peut aussi préciser une classe parente (héritage de classe au sens classique) et un ou plusieurs modules à incorporer.

Les méthodes sont fournies au travers d'un objet anonyme dont elles sont les propriétés. La méthode `initialize` a un rôle particulier : c'est le « constructeur » de la classe. Contrairement à ce qui avait cours en version 1.5, cette méthode est désormais optionnelle : si votre classe ne fait rien de spécial au moment de son instantiation, inutile de la déclarer : elle sera remplacée par la fonction générique `Prototype.emptyFunction`.

Lorsqu'on a une classe sous la main, on peut déterminer sa classe parente avec sa propriété `superclass` (`null` si la classe n'a pas de classe parente explicite) et lister ses classes filles avec sa propriété `subclasses` (tableau vide si aucun héritage n'existe pour le moment). Il est donc facile d'établir, à partir d'une classe, toute la hiérarchie de classes à laquelle elle appartient. Nous verrons un exemple à la section suivante.

Toute instance d'une classe produite par `Class.create` peut par ailleurs indiquer son constructeur, c'est-à-dire la classe dont elle est une instance, avec sa propriété `constructor`, ce qui est cohérent avec les classes natives de JavaScript, dont les instances ont cette même propriété.

Voici un premier exemple tout simple :

```
var ImageLoader = Class.create({
  initialize: function() {
    this.images = [];
    this.imageURLs = $A(arguments);
    this.loaded = false;
    this.startLoading();
  },
  startLoading: function() {
    // ...
  }
});
new ImageLoader('/i/banner.png', '/i/preview.jpg'...);
// ...
if (img.loaded) ...
```

Lorsqu'on souhaite incorporer un module à une classe, on peut le faire soit à la définition de celle-ci, soit *a posteriori* avec `addMethods` :

```
var ByteSequence = Class.create(Enumerable, {
  _each: function() {
    // ...
  },
  ...
});
```

Dans le code ci-dessus, `Class.create` détecte que son premier argument n'est pas une classe (il ne s'agit pas d'une fonction constructeur) et le traite donc comme un module, en appelant en interne `addMethods` avec lui. On peut d'ailleurs avoir autant de modules qu'on veut :

```
var MyClass = Class.create(Enumerable, MyCoolModule, {
  ...
});
```

Lorsqu'on souhaite incorporer un module plus tard (par exemple depuis un plug-in chargé après le code de base), on utilise directement `addMethods` :

```
MyClass.addMethods(MyCoolModule);
```

L'héritage et la surcharge de méthode

Comme nous l'avons vu plus haut, pour hériter d'une classe lorsqu'on en crée une autre, il suffit de passer la classe parente comme premier argument à `Class.create` :

```
var Animal = Class.create({ ... });
var Mammal = Class.create(Animal, { ... }); // Mammifère
var animal = new Animal(...);
var mammal = new Mammal(...);
Animal.superclass // => null
Animal.subclasses // => [Mammal]
animal.constructor // => Animal
Mammal.superclass // => Animal
Mammal.subclasses // => []
mammal.constructor // => Mammal
```

On peut bien sûr passer une classe parente *et* des modules à incorporer :

```
var Sequence = Class.create({ ... });
var ByteSequence = Class.create(Sequence, Enumerable, { ... });
```

Là où ça devient vraiment sympathique, c'est que Prototype nous permet de surcharger des méthodes dans une classe fille tout en récupérant une référence sur la version héritée, afin de faire l'équivalent d'un appel **inherited** (Delphi) ou **super** (Java, C#). Il suffit pour cela de démarrer la signature de notre surcharge par le paramètre spécial `$super` :

```
var Animal = Class.create({
  initialize: function(name) { this.name = name; },
  speak: function(what) { alert(this.name + ' dit : ' + what); }
});
var Cat = Class.create(Animal, {
  speak: function($super, what) { $super('Meow ! ' + what); }
});
var truc = new Animal('Truc');
var felix = new Cat('Felix');

truc.speak('Mwaaa');
// => 'Truc dit : Mwaaa'

felix.speak("J'ai faim !");
// => 'Felix dit : Meow ! J'ai faim !'
```

Vous remarquerez au passage que `$super` a directement le bon *binding* : même si c'est une référence de fonction qui vous est passée, elle ne perd pas le sens de `this` pour son exécution.

Remarquez aussi que Cat a hérité la méthode `initialize` de `Animal` : en ne déclarant pas son propre « constructeur », il n'a pas récupéré `Prototype.emptyFunction` à la place, mais bien le `initialize` hérité de `Animal`, puisque sa propriété `name` est bien initialisée à la construction.

Manipulation d'éléments

Attaquons maintenant les objets dédiés à la manipulation du contenu de la page.

Nous verrons d'abord les manipulations valables pour tout élément du document, avec `Element`. Après un rapide détour par `Selector` (sans lequel la fonction globale `$$` n'existerait pas), nous verrons les objets dédiés aux formulaires et la gestion unifiée des événements.

Element, votre nouveau meilleur ami

L'objet `Element` est un espace de noms pour de nombreuses méthodes destinées à manipuler des éléments du DOM. En raison de leur nombre, je les ai classés par thème pour qu'on s'y retrouve plus facilement.

Element.Methods et les éléments étendus

Techniquement, ces méthodes sont en réalité définies dans un module `Element.Methods`, lequel est ensuite incorporé à `Element`. La raison de cette séparation, qui peut d'abord sembler superflue, réside dans la notion d'**élément étendu**, que nous avons déjà vue en début de chapitre.

En isolant les méthodes d'extension des quelques méthodes « administratives » internes à `Element`, Prototype est capable de les injecter si nécessaire dans les éléments du DOM qui vous sont retournés par des fonctions comme `$` ou `$$`.

Le résultat net en est que dans la vaste majorité des cas, les deux syntaxes suivantes sont strictement équivalentes :

SYNTAXE

```
Element.methode(votreElement, arg...)
votreElement.methode(arg...)
```

C'est la « méthodisation » que nous avons évoquée plus haut en étudiant les ajouts que Prototype fournit aux fonctions.

Quelques petits exemples pour fixer les idées :

```
Element.hide('monId');
$('monId').hide();
monElement.hide();
Element.update(monElement, '<strong>Génial !</strong>');
$('monId').update('<strong>Génial !</strong>');
monElement.update('<strong>Génial !</strong>');
```

Quand ça ne veut pas...

Il reste une question en suspens : dans quels cas cet accès direct (`monElement.méthode`) n'est-il pas disponible ? Il faut que les conditions suivantes soient réunies :

- 1 Votre navigateur ne propose pas d'objet `prototype` pour les éléments issus du DOM.
- 2 Vous travaillez sur un élément que vous n'avez pas récupéré au travers d'une méthode documentée dans ce chapitre comme renvoyant un élément garanti étendu (`eltÉtendu` dans les blocs de syntaxe).

À l'heure où j'écris ces lignes, Firefox, Camino, Mozilla, Safari et Konqueror proposent un objet `prototype` pour les éléments du DOM. La condition n°1 est donc remplie uniquement sur MSIE et Opera !

Le principal cas de figure qui remplit la condition n°2 est la traversée « à l'ancienne » du DOM. Vous récupérez par exemple un élément avec `$('sonId')`, puis vous commencez à naviguer avec `firstChild`, `nextSibling`, `parentNode`, etc. Cette navigation ne passe bien sûr pas par Prototype, et si la condition n°1 est remplie, les éléments retournés ne sont pas étendus.

Dans la pratique, ceci n'est pas un problème. Vous connaissez votre code, et si vous utilisez une traversée du DOM, vous êtes au courant. Dans un tel contexte, vous n'avez qu'à utiliser la version indirecte (`Element.méthode(elt)`) des méthodes ou passer d'abord votre élément à `$()` pour ensuite utiliser les méthodes étendues.

Le reste du temps, vous pouvez normalement utiliser sans crainte la version directe, qui est en effet plus concise, plus élégante, et globalement plus naturelle.

Néanmoins, par souci de « garantie de fonctionnement », les sections qui suivent documenteront les syntaxes indirectes. À vous de garder à l'esprit qu'une version directe est le plus souvent disponible.

Valeur de retour des méthodes

Toutes les méthodes de `Element` (ainsi que de `Form` et `Form.Element`) qui modifient l'élément visé *renvoient cet élément étendu*. Cela permet d'enchaîner les modifications, par exemple :

```
| $('sidebar').addClassName('selected').show();
```

Enfin manipuler librement les attributs : `hasAttribute`, `readAttribute` et `writeAttribute`

Si vous avez déjà développé une portion significative de code manipulant les attributs (les propriétés DOM correspondant aux attributs HTML) en essayant de fonctionner sur tous les navigateurs, vous savez à quel point cela peut être un cauchemar.

D'une part, certains attributs HTML ont des noms qui sont des mots réservés en JavaScript, comme `for` et `class`, qui deviennent `htmlFor` et `className`. D'autre part, certains attributs ont des représentations spéciales au niveau DOM, comme les attributs « drapeau » tels que `checked`, `disabled`, `readonly` et `selected`. En XHTML, leur seule valeur autorisée est leur propre nom, mais dans le DOM, on obtient tantôt `true` et `false`, tantôt leur nom, tantôt `null`, tantôt `yes` ou `no`... Et que dire lorsqu'on doit leur affecter une valeur plutôt que les lire ?

Heureusement, Prototype remplit pleinement son rôle de « lisiteur d'incompatibilités » sur ce point (comme il le fait avec la manipulation des styles au travers de `getStyle` et `setStyle`, que nous verrons plus loin).

SYNTAXE

```
Element.readAttribute(elt, nomAttribut) -> valeur
Element.writeAttribute(elt, nomAttr, valeur) -> eltÉtendu
Element.writeAttribute(elt, { attr: val... }) -> eltÉtendu
Element.hasAttribute(elt, nomAttr) -> booléen
```

La méthode `readAttribute` accepte aussi bien les noms HTML que DOM pour les attributs, et garantit un type de retour homogène et conforme aux spécifications DOM niveau 2 HTML (par exemple, en renvoyant leur propre nom ou `null` pour les attributs « drapeau »). Elle est donc bien supérieure à la méthode DOM `getAttribute`... Si l'attribut n'est pas présent, elle renvoie `null`.

Réciproquement, `writeAttribute` est tout aussi libérale sur les noms, et accepte les diverses possibilités de valeur pour les attributs « drapeau ». En plus de ça, elle permet de définir plusieurs attributs d'un coup en passant un *hash* d'attributs au lieu d'un nom et d'une valeur. Là aussi, le `setAttribute` natif ne fait plus le poids...

Enfin, `hasAttribute` est un peu particulière : cette méthode est normalement fournie par le DOM, mais comme souvent, MSIE ne la prend pas toujours en charge (MSIE 6 et 7 notamment), aussi Prototype la garantit-il.

Élément es-tu là ? `hide`, `show`, `toggle` et `visible`

Quatre méthodes gèrent la visibilité de l'élément. Elles utilisent toutes la propriété CSS `display`, qu'elles placent à '`none`' pour masquer l'élément, et qu'elles désactivent (en la ramenant à '`'`) pour afficher l'élément. Ceci suppose deux choses :

- 1 Les éléments concernés seront retirés du flux du document lorsqu'ils sont masqués, engendrant un *reflow* s'ils faisaient partie du flux (un élément positionné de façon absolue ou fixe n'en fait pas partie, par exemple).
- 2 Si vous souhaitez masquer l'élément d'entrée de jeu, vous devez le faire avec un **attribut style dans le document**, et non avec la feuille CSS. C'est le seul cas de figure dans ce livre qui justifie une légère intrusion de l'aspect dans le contenu.

SYNTAXE

```
Element.hide(elt) -> eltÉtendu
Element.show(elt) -> eltÉtendu
Element.toggle(elt) -> eltÉtendu
Element.visible(elt) -> booléen
```

Les noms me semblent parfaitement explicites. Les méthodes `hide` et `show` masquent l'élément en exigeant un `display` à '`none`' ou en retirant cette surcharge, tandis que `toggle` bascule de l'un à l'autre.

La méthode `visible` est en réalité un prédicat, qui indique si l'élément est visible ou non. Elle le considère invisible si sa propriété `display` est à '`none`'. Il est dommage que Prototype ne préfixe pas ses méthodes prédictives par un '`is`', mais la version 1.6.1 pourrait commencer à remédier à ce choix initial un peu malheureux...

Gestion du contenu : `cleanWhitespace`, `empty`, `remove`, `replace`, `update`, `insert`, `wrap` et la syntaxe constructeur

Prototype fournit nombre de méthodes visant à modifier ou remplacer le contenu d'un élément, voire à remplacer l'élément lui-même, ou carrément le retirer du DOM.

SYNTAXE

```
Element.cleanWhitespace(elt) -> eltÉtendu
Element.empty(elt) -> booléen
Element.insert(elt, contenuUnique) -> eltÉtendu
```

```
Element.insert(elt, { position: contenu... }) -> eltÉtendu
Element.remove(elt) -> eltÉtendu
Element.replace(elt, contenu) -> eltÉtendu
Element.update(elt, contenu) -> eltÉtendu
Element.wrap(elt[, enrobeur][, attributs]) -> enrobeurÉtendu
new Element(balise[, attributs]) -> eltÉtendu
```

La méthode `cleanWhitespace` supprime tout noeud fils textuel « vide » (constitué uniquement de blancs). C'est une sorte d'épuration du DOM qui permet de simplifier les algorithmes de traversée (`firstChild`, `lastChild` ou `nextSibling` sont généralement des éléments comme on pourrait s'y attendre, plutôt que des noeuds textuels vides dus à l'indentation d'un code source).

La méthode `empty` est un prédictat indiquant si l'élément est vide, c'est-à-dire ne contenant au plus que du *whitespace*.

Comme son nom le laisse supposer, `remove` retire l'élément du DOM. C'est juste une simplification du sinueux `elt.parentNode.removeChild(elt)`. Mais en accès direct, avouez que `elt.remove()`, c'est plus sympa !

Les deux méthodes `replace` et `update` ne diffèrent que sur un point : `update` remplace le *contenu* de l'élément, tandis que `replace` remplace *l'élément* lui-même ! La différence est cependant de taille pour vos algorithmes. En pratique, `update` est plus fréquemment utilisée.

Les deux interprètent le contenu comme suit :

- Si la valeur passée fournit une méthode `toElement()`, celle-ci est d'abord utilisée pour obtenir un contenu alternatif ; cela ouvre la porte à l'utilisation d'objets inattendus comme arguments de `replace` et `update`, du moment qu'ils respectent ce « protocole de conversion ».
- Si le contenu à ce stade est un élément DOM, il est utilisé directement.
- Dans le cas contraire, on examine si le contenu fournit une méthode `toHTML()`, dont le résultat est alors utilisé (second protocole de conversion). Sinon, on se garantit simplement contre un contenu de type `null` ou `undefined`, qui sera considéré comme un fragment HTML vide
- À ce stade final, on a un contenu textuel sous la main, qui est traité comme un fragment HTML. Les fragments de script sont retirés avant insertion dans le DOM, puis ces mêmes scripts sont exécutés à part, juste après l'insertion (à 10 ms d'écart, vous ne risquez pas de sentir la différence). L'idée est que ces fragments de script ne sont pas censés persister dans le DOM, mais qu'il peut être intéressant d'injecter tant du contenu que du comportement dans un document (notamment pour les réponses Ajax).

La méthode `insert` a remplacé et amélioré, dans Prototype 1.6, l'ancien espace de noms `Insertion`. Elle traite ses contenus exactement de la même manière que `replace` et `update`, manière décrite à l'instant. Elle prend en charge deux syntaxes :

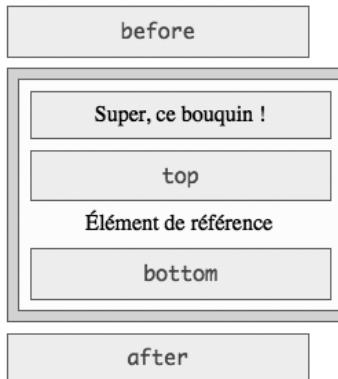
SYNTAXE

```
Element.insert(elt, contenuUnique) -> eltÉtendu
Element.insert(elt, { position: contenu... }) -> eltÉtendu
```

Il existe quatre positions pour l'insertion : `before`, `top`, `bottom` et `after`. La figure 4-2 illustre leurs emplacements respectifs.

Figure 4-2

Les différentes positions d'insertion



Une insertion à contenu unique est équivalente à une insertion `bottom` :

```
Element.insert(elt, 'demo')
// équivalent à :
Element.insert({ bottom: 'demo' })
```

Pour être perçu comme un contenu unique et non une série de contenus positionnés, l'argument doit être une chaîne de caractères, un nombre, un élément DOM, ou un objet possédant l'un des deux protocoles de conversion `toElement()` ou `toHTML()`. Tout autre argument sera traité comme une liste de contenus positionnés par les quatre noms `before`, `top`, `bottom` et `after`.

La méthode `wrap` permet également d'insérer un élément, mais « autour » d'un élément existant, qui devient en fait l'élément fils de l'élément d'enrobage, lequel prend la place, dans le DOM, de l'élément d'origine. On peut préciser le contenu d'enrobage sous forme d'une balise optionnelle (par défaut `div`), elle-même suivie d'une série d'attributs HTML optionnels (dont on peut se servir pour affecter directement l'élément d'enrobage fraîchement créé, par exemple en lui affectant un style, des classes CSS, un index de tabulation, etc.)

La figure 4-2 est ainsi partie du HTML suivant :

```
<div id="ref">Élément de référence</div>
```

Puis on a exécuté le code que voici :

```
// Syntaxe à contenu unique
$('ref').insert('<div class="element bottom">' +
  '<code>bottom</code></div>');
// Syntaxe à contenus multiples
$('ref').insert({
  before: '<div class="element before"><code>before</code></div>',
  top: '<div class="element top"><code>top</code></div>',
  after: '<div class="element after"><code>after</code></div>'
});
$('ref').wrap({ className: 'element wrap' });
// Un peu de scripting à la volée ?
$('ref').insert({ top:
  '<div class="element top" id="scripted">Such a nice book</div>' +
  '<script type="text/javascript">' +
  '$("scripted").update("Super, ce bouquin !")' +
  '</script>' });
```

Pour en terminer avec l'insertion, sachez que depuis Prototype 1.6, Element n'est pas seulement un espace de noms, mais aussi une fonction (utilisée d'ailleurs en interne par wrap, entre autres). Elle porte d'ailleurs la même signature que wrap, et permet de créer rapidement un élément et ses attributs HTML. Si vous avez des besoins de création plus étendus, il faudra soit passer par un fragment HTML textuel, soit utiliser le Builder de script.aculo.us.

```
var field = new Element('input', { name: 'login',
  id: 'edtReqLogin', 'class': 'field', tabIndex: 42 });
```

Styles et classes : addClassName, getOpacity, getStyle, hasClassName, match, removeClassName, setOpacity, setStyle et toggleClassName

Nous disposons de nombreuses méthodes pour gérer les classes CSS et les propriétés CSS individuelles.

SYNTAXE

Element.addClassName(elt, className) -> elt	Étendu
Element.getOpacity(elt) -> opacité (0 à 1)	
Element.getStyle(elt, style) -> valeur	
Element.hasClassName(elt, className) -> booléen	

```
Element.match(elt, singleSelector) -> booléen  
Element.removeClass(elt, className) -> eltÉtendu  
Element.setOpacity(elt, opacité) -> eltÉtendu  
Element.setStyle(elt, styleText) -> eltÉtendu  
Element.setStyle(elt, { prop1: val1[...] }) -> eltÉtendu  
Element.toggleClassName(elt, className) -> eltÉtendu
```

Comme vous le savez, un même élément peut avoir plusieurs classes CSS associées, en séparant leurs noms par des espaces. Afin de vous éviter d'avoir à gérer les manipulations de texte, vous disposez donc des méthodes `addClassName`, `removeClassName`, `hasClassName` et `toggleClassName`. Leurs noms parlent d'eux-mêmes...

Vous pouvez par ailleurs manipuler confortablement les propriétés CSS individuelles de votre élément au travers des méthodes `getStyle` et `setStyle`. Derrière leur apparence simpliste, elles réalisent un travail complexe.

Ainsi, `getStyle` ne cherche pas qu'à vous renvoyer la *valeur spécifiée* de la propriété. Si une telle valeur existe, elle vous la renvoie bien entendu. Dans le cas contraire, elle se débrouille avec les possibilités du navigateur (DOM niveau 2 Style ou API propriétaire) pour extraire la *valeur calculée* de la propriété, qui peut par exemple résulter d'un style par défaut ou de l'héritage. Si la prise en charge du navigateur fait que tout ceci aboutit néanmoins à '`'auto'`', `getStyle` renvoie `null`. Sinon, elle renvoie la valeur calculée. Enfin, notez que le nom de propriété passé à `getStyle` peut être indifféremment celui de la recommandation CSS (par exemple, `float` ou `border-left-color`) ou celui de la propriété DOM correspondante (resp. `cssFloat` et `borderLeftColor`).

Si ces notions de valeur spécifiée, valeur calculée, cascade ou héritage sont floues pour vous, n'hésitez pas à lire tranquillement le début de l'annexe B, qui les décrit avec précision.

La méthode `setStyle` utilise une syntaxe plus évoluée pour son deuxième argument, afin de permettre la définition d'un nombre quelconque de propriétés en un seul appel. On lui passe soit un fragment de définition textuelle de style (exactement le genre qu'on passerait comme valeur d'un attribut `style=` en HTML), soit un objet anonyme dont les propriétés ont le nom DOM (**pas** le nom CSS...) de la propriété CSS voulue, avec leurs valeurs. Exemples :

```
Element.setStyle('indicator', 'height: 2em');  
$('notif').setStyle({ backgroundColor: '#fdd', color: 'maroon' });
```

Prototype fait des merveilles avec la gestion de l'opacité. Comme vous le savez peut-être, MSIE 6 ne prend pas en charge la propriété CSS `opacity`. C'est bien gênant pour les développeurs web, qui doivent alors utiliser une syntaxe de filtre CSS

propriétaire. Par exemple, au lieu de `opacity: 0.5`, il leur faudrait indiquer `filter: alpha(opacity=50)`.

Prototype lisse ces différences en écriture comme en lecture, au travers de `getStyle` et `setStyle`, mais la manipulation de l'opacité est tellement courante de nos jours, avec des systèmes de type Lightbox par exemple, que les éléments étendus disposent de deux méthodes dédiées : `getOpacity` et `setOpacity`, qui renvoient et prennent directement une valeur standardisée (de 0 pour la transparence à 1 pour l'opacité totale).

La méthode `match` accepte un sélecteur unique (donc sans espaces) et renvoie `true` si l'élément satisfait ce sélecteur.

Petite note pour finir : l'objet `Element.ClassNames` et la méthode `classNames()` associée ont été dépréciés, dans la mesure où ils n'apportaient guère de plus-value mais étaient pénalisants en performance.

Les copains d'abord : `ancestors`, `childElements`, `descendantOf`, `descendants`, `firstDescendant`, `immediateDescendants`, `previousSiblings`, `nextSiblings`, `siblings`

Ces méthodes permettent de récupérer les noeuds « autour » de l'élément, que ce soit ceux en proximité immédiate ou ceux plus lointains.

SYNTAXE

```
Element.ancestors(elt) -> [eltÉtendu...]
Element.descendantOf(elt, eltRef) -> booléen
Element.descendants(elt) -> [eltÉtendu...]
Element.firstDescendant(elt) -> eltÉtendu
Element.childElements/immediateDescendants(elt) -> [eltÉtendu...]
Element.nextSiblings(elt) -> [eltÉtendu...]
Element.previousSiblings(elt) -> [eltÉtendu...]
Element.siblings(elt) -> [eltÉtendu...]
```

La lignée (éléments ancêtres) est renvoyée par `ancestors` depuis le noeud père jusqu'au plus lointain ancêtre. Le tableau renvoyé par `descendants` suit l'ordre du document et renvoie toute la grappe d'éléments sous l'élément indiqué. Inversement, si on veut tester que l'élément courant est descendant d'un autre, on peut utiliser `descendantOf` (qui était autrefois bien mal nommé `childOf`). Lorsqu'on veut juste ses éléments fils, on utilise `immediateDescendants` (`children` aurait été préférable mais existe déjà, avec une sémantique différente, sur Safari... En revanche, l'alias « `childElements` », que je préfère, existe). Les méthodes `nextSiblings` et `siblings`

suivent l'ordre du document, tandis que `previousSiblings` suit l'ordre inverse (en s'éloignant de l'élément d'origine).

Bougez ! adjacent, down, next, previous, select et up

Pour se déplacer le long des axes (éléments frères, ancêtres, descendants), Prototype fournit 6 méthodes qui se basent sur une sélection CSS.

SYNTAXE

```
Element.adjacent(elt, expression) -> [eltÉtendu...]
Element.down(elt[, expression][, index]) -> eltÉtendu
Element.next(elt[, expression][, index]) -> eltÉtendu
Element.previous(elt[, expression][, index]) -> eltÉtendu
Element.select(elt, expression) -> [eltÉtendu...]
Element.up(elt[, expression][, index]) -> eltÉtendu
```

Le sens de déplacement est évident ; par exemple, `down` utilise `firstDescendant()` ou `descendants()`, tandis que `previous` passe par `previousSiblings()`. Les éléments renvoyés le sont dans l'ordre prévu par les méthodes correspondantes dans la section précédente.

L'argument `expression` fournit un sélecteur CSS unique (sans espaces). Pour `up`, `down`, `previous` et `next` il est optionnel, tandis que `index` permet d'indiquer la position de la correspondance sur laquelle on souhaite arriver (démarre à zéro). On peut utiliser l'un, l'autre ou les deux. Prototype optimise son algorithme en fonction de la signature retenue à l'exécution.

Deux nouvelles méthodes ont fait leur apparition en 1.6. Tout d'abord `select`, qui remplace l'ancienne méthode `getElementsBySelector` (on a gagné au change !) et renvoie l'ensemble des éléments descendants satisfaisant le sélecteur CSS. On s'en sert énormément, et il est considéré plus propre de l'utiliser sur un élément racine que de passer par `$$` et démarrer l'expression par un sélecteur d'ID :

```
// Moyen propre, vu qu'on ne peut pas factoriser/réutiliser
// l'expression de sélection :
$$('#item .nav[tabindex]')

// Plus modulaire, plus factorisable, plus sympa :
$('item').select('.nav[tabindex]')
```

L'autre nouvelle méthode est `adjacent`, qui établit une sélection CSS sur les `siblings()`, les éléments frères.

Voyons quelques exemples d'appels. Supposons l'arborescence suivante :

```
<div id="sidebar">
  <ul id="nav">
    <li id="first">...</li>
    <li>...</li>
    <li class="selected">...</li>
    <li id="demo">...</li>
  </ul>
  <ul id="menu">
    ...
  </ul>
```

Voici maintenant quelques utilisations et leurs résultats commentés :

Tableau 4-1 Tableau 4.1 Invocations de up, down, previous, next, adjacent et select

Code	Élément résultat	Commentaires
<code>\$('nav').up() \$('menu').up('div')</code>	<code><div id="sidebar"></code>	Montée d'un cran sans contrainte aucune, ou jusqu'au premier ancêtre div.
<code>\$('sidebar').down() \$('sidebar').down('ul') \$('menu').previous()</code>	<code><ul id="nav"></code>	Descente d'un cran depuis sidebar, descente jusqu'à un ul, ou élément précédent menu...
<code>\$('sidebar').down(1) \$('sidebar').down('li')</code>	Premier <code>...</code>	Descente de deux crans (position 1, on démarre à zéro) depuis sidebar, ou descente jusqu'à un li.
<code>\$('sidebar').down(2) \$('sidebar').down('li', 2) \$('sidebar').down('li').next('li')</code>	Deuxième <code>...</code>	Descente de trois crans depuis sidebar, voire avec contrainte li en plus, ou prochain li après le premier li descendant de sidebar.
<code>\$('sidebar').down('li.selected')</code>	<code><li class="selected"> ...</code>	Descente jusqu'au premier li ayant une classe CSS selected.
<code>\$('sidebar').down('ul').next()</code>	<code><ul id="menu"></code>	Élément suivant le premier ul descendant de sidebar.
<code>\$('nav').down('.selected') .adjacent('*[id]')</code>	<code>[<li id="first">..., <li id="demo">...]</code>	Éléments frères du premier li descendant avec une classe sidebar, qui auraient un attribut id.
<code>\$('sidebar').select(':not([id])')</code>	<code>[..., <li class="selected"> ...]</code>	Descendants de sidebar qui n'ont pas d'attribut id.

Remarquez notamment que `next` et `previous` éliminent le problème des nœuds texte vides issus de l'indentation, problème que nous avons détaillé au chapitre 3.

Positionnement : `absolutize`, `clonePosition`, `cumulativeOffset`, `cumulativeScrollOffset`, `getDimensions`, `getHeight`, `getOffsetParent`, `getWidth`, `makePositioned`, `positionedOffset`, `relativize`, `undoPositioned` et `viewportOffset`

La gestion des positions et dimensions est un des plus gros casse-tête qu'affrontent développeurs web et auteurs de bibliothèques JavaScript. Ce seul domaine représente 90 % du travail sur Prototype 1.6.0.3, soit des mois-homme de travail. D'un navigateur à l'autre, rien ne va plus...

Prototype fournit donc une moisson de méthodes dédiées à ce sujet, largement exploitées ensuite par script.aculo.us pour ses effets visuels et la mise en œuvre du glisser-déplacer.

SYNTAXE

```
Element.absolutize(elt) -> eltÉtendu
Element.clonePosition(elt, source[, options]) -> eltÉtendu
Element.cumulativeOffset(elt) -> offset
Element.cumulativeScrollOffset(elt) -> offset
Element.getDimensions(elt) -> { width: Number, height: Number }
Element.getHeight(elt) -> Number
Element.getOffsetParent(elt) -> eltÉtendu
Element.getWidth(elt) -> Number
Element.makePositioned(elt) -> eltÉtendu
Element.positionedOffset(elt) -> offset
Element.relativize(elt) -> eltÉtendu
Element.undoPositioned(elt) -> eltÉtendu
Element.viewportOffset(elt) -> offset
```

Les méthodes `getHeight` et `getWidth` renvoient la hauteur et la largeur en pixels de l'élément. La méthode `getDimensions()` renvoie les deux au travers d'un objet doté de deux propriétés : `width` et `height`, deux nombres exprimés là encore en pixels.

`makePositioned` s'assure que l'élément devient un conteneur de positionnement (un référentiel de coordonnées en positionnement absolu pour ses éléments descendants). Pour cela, elle examine la propriété CSS `position` de l'élément : si elle est indéfinie, ou explicitement spécifiée à `static` (sa valeur par défaut), elle la passe en `relative` après avoir sauvégardé son ancienne valeur pour restauration.

Cette restauration est justement l'affaire de `undoPositioned`, qui remet la propriété CSS `position` à son ancienne valeur. Cette méthode n'a donc de sens que suite à un `makePositioned`.

Pour basculer un élément en positionnement absolu sans modifier sa position apparente à l'écran, on a justement la méthode `absolutize`. Celle-ci avait d'ailleurs des soucis, avant la 1.6.0.3, lorsque l'élément présentait de l'espacement interne (*padding*) ou une bordure... Pour annuler la conversion ainsi réalisée, on peut appeler `relativize`. (mais on n'a toujours pas trouvé de raison de le faire). Attention, ces méthodes pourraient très prochainement être renommées dans un souci de clarté en `makeAbsolute` et `undoAbsolute`.

Prototype équipe par ailleurs les éléments DOM avec une série de méthodes qui calculent leur `offset` (la position de leur coin supérieur gauche) par rapport à différents référentiels de coordonnées. Voici un rapide examen de ces méthodes ; la figure 4-3 essaie de rendre tout ça plus clair.

- `cumulativeOffset` calcule la position à l'intérieur du document lui-même.
- `cumulativeScrollOffset` donne la valeur consolidée des décalages horizontal et vertical dus aux différents niveaux conteneurs autour de l'élément, lesquels auraient défilé.
- `getOffsetParent` renvoie le **conteneur de positionnement** de l'élément. C'est une notion centrale en positionnement CSS : lorsqu'un élément est positionné en absolu, l'origine de cette position (la coordonnée (0,0)) est le coin supérieur gauche de l'espacement interne éventuel de son conteneur de positionnement. Pour être un tel conteneur, un élément doit être *positionné*, c'est-à-dire avoir une propriété CSS `position` valant `absolute`, `relative` ou `fixed`. Faute d'un tel élément ancêtre, le conteneur de positionnement par défaut est `document.body`.
- `positionedOffset` donne la position à l'intérieur du conteneur de positionnement, justement.
- `viewportOffset`, enfin, donne la position à l'intérieur du *viewport*, la zone de la fenêtre au sein de laquelle est affiché le document. Prototype propose d'ailleurs diverses fonctions autour de la notion de *viewport*, que nous verrons plus tard.

Dans le bloc de syntaxe, j'ai décrit le type de retour des méthodes `...Offset` comme un « offset ». Il s'agit en fait d'un tableau de deux éléments, dont le premier est la position horizontale et le second la position verticale, mais ces informations sont également accessibles au travers de deux propriétés dédiées sur ce tableau : `left` et `top`. À chacun son style !

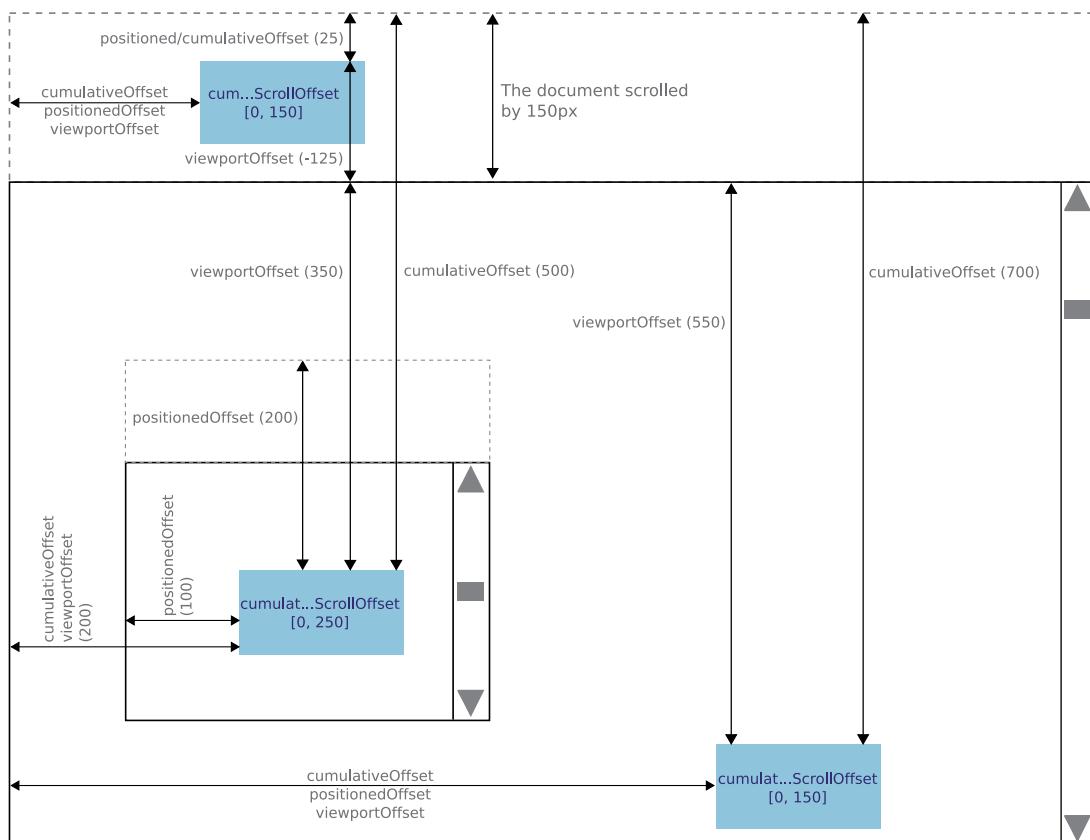


Figure 4–3 Les propriétés d'offset et leurs relations respectives

La méthode `clonePosition` permet quant à elle de cloner les propriétés positionnelles (position et taille) d'un élément sur l'élément courant, ce qui peut s'avérer fort pratique. Elle autorise six options :

- `setLeft`, `setTop`, `setWidth` et `setHeight` indiquent quelles propriétés cloner, et valent toutes `true` par défaut.
- `offsetLeft` et `offsetTop`, des valeurs en pixels qui sont à zéro par défaut, permettent de décaler l'élément par rapport à la position exacte de l'élément de référence. On peut décaler horizontalement (e.g. zone de notification d'erreur pour un champ de formulaire), verticalement (e.g. zone de complétion pour un champ de saisie) ou les deux (e.g. ombre portée créée en ajustant aussi `z-index`).

Une petite remarque pour terminer : nombre de ces méthodes se trouvaient autrefois, parfois sous un nom différent, dans l'espace de noms `Position`, qui a été déprécié avec la version 1.6.

Défilement et troncature : makeClipping, scrollTo, undoClipping

Voici trois méthodes gérant le défilement et la troncature de contenu.

SYNTAXE

```
Element.scrollTo(elt) -> eltÉtendu  
Element.makeClipping(elt) -> eltÉtendu  
Element.undoClipping(elt) -> eltÉtendu
```

La méthode `scrollTo` fait si besoin défiler la vue du navigateur de façon à ce que l'élément y soit visible. C'est très pratique suite à l'insertion dynamique d'un élément par exemple, pour s'assurer que l'internaute voie bien la modification. Et cela nous évite de devoir recourir à des bidouilles navrantes d'ancre dans l'URL de la page.

Les méthodes `makeClipping` et `undoClipping` ont la même relation que `makePositioned` et `undoPositioned`. Il s'agit ici de déterminer si, lorsqu'une boîte dimensionnée a trop de contenu, celui-ci doit étirer son conteneur ou être partiellement masqué, la boîte conservant sa taille.

Techniquement, `makeClipping` sauvegarde la valeur actuelle de la propriété CSS `overflow`, et la passe à `hidden`. `undoClipping` restaure.

Ajouter des méthodes aux éléments étendus

Peut-être voudrez-vous un jour ajouter vos propres extensions aux éléments étendus par Prototype. Rien de plus facile en vérité : il vous suffit de les regrouper dans un module et d'utiliser `Element.addMethods` (méthode statique). Suivant que vous souhaitez étendre toutes les balises ou seulement certaines, vous les préciserez au besoin.

SYNTAXE

```
Element.addMethods({ m1: f1... })  
Element.addMethods(balise, { m1: f1... })  
Element.addMethods([balise...], { m1: f1... })
```

Les méthodes sont fournies sous la forme d'un objet dont elles sont les propriétés, par exemple un module ; vous retrouvez là le mode de fonctionnement de `Class.create` et `Class.addMethods`, par exemple. Mais avant ce module, on peut préciser une balise voire un tableau de balises, auquel cas l'extension ne sera opérante que sur les balises ainsi désignées, au lieu de s'appliquer à tous les éléments.

Voici un petit exemple tiré d'un de mes récents travaux, simplifié toutefois pour éviter d'introduire une complexité hors de propos ici (sa version officielle est plus

optimisée, mais plus dure à lire pour les néophytes...), qui trouvera d'ailleurs peut-être son chemin dans la 1.6.0.3 pendant l'impression de ce livre...

```
var ImageExtensions = {
    isLoaded: function(elt) {
        return elt.readyState == 'complete' || elt.complete ||
        ('undefined' == typeof elt.naturalWidth || 0 != elt.naturalWidth);
    }
};
Element.addMethods('IMG', ImageExtensions);
```

Et voilà, tous les éléments `` accédés par la suite de façon étendue seront dotés de cette méthode `isLoaded()`. Remarquez bien que vos méthodes doivent être définies de façon plus « procédurale », avec l'élément sujet en premier argument.

Quelques détails pour finir...

Peut-être vous arrive-t-il d'utiliser les constantes de type de nœud DOM normalement fournies par l'interface `Node`, par exemple `Node.ELEMENT_NODE`, pour tester les valeurs de la propriété `nodeType`. Le souci évidemment, c'est que sur MSIE, l'interface n'existe pas toujours... Sachez donc que Prototype la garantit, avec ses douze constantes de type.

Il reste encore deux petites méthodes étendues que je voulais vous signaler, mais qui n'entraient guère dans les autres catégories :

SYNTAXE

```
Element.inspect(elt) -> chaîne
Element.identify(elt) -> ID
```

D'abord, les éléments DOM sont dotés d'une méthode `inspect`, qui leur permet de participer au système d'inspection mis en place par Prototype un peu partout ailleurs (types natifs et méthode statique `Object.inspect`), déjà étudié dans ce chapitre. Cette méthode est d'ailleurs intéressante puisqu'elle renvoie une représentation de la balise ouvrante avec ses éventuels attributs `id` et `class`, ce qui est bien utile (l'ordre des deux n'étant toutefois pas garanti).

Par exemple :

```
var par = new Element('p', { id: 'demo', className: 'funny' });
par.inspect()
// => '<p id="demo" class="funny">'
```

Il arrive également régulièrement qu'on doive passer un élément à un morceau de code qui exigera pour fonctionner que l'élément soit doté d'un identifiant (attribut `id`). Or, les identifiants d'un document doivent être uniques... Pour répondre à ce besoin, Prototype fournit une méthode `identify` qui associera à l'élément, s'il n'a pas déjà un identifiant, un identifiant synthétique au format `anonymous_element_nombre`, le nombre démarrant à 1. L'unicité est systématiquement vérifiée, et tant qu'elle n'est pas avérée, le suffixe numérique augmente, Prototype le gardant en mémoire pour éviter de repartir de 1 à chaque tentative... Soit le HTML suivant :

```
<body>
  <p id="demo">Élodie</p>
  <p>Aurore</p>
  <p>Hélène</p>
  <div id="anonymous_element_2">Thifaine</div>
</body>
```

On aurait le comportement suivant :

```
$(document).invoke('identify')
// => ['demo', 'anonymous_element_1', 'anonymous_element_3']
```

La gestion du `viewport`

Depuis sa version 1.6, Prototype permet d'examiner le `viewport`, la zone de la fenêtre où s'affiche effectivement le document. Pouvoir en déterminer les dimensions, notamment, est essentiel pour réaliser des fonctionnalités comme le défilement automatique façon Gmail, et bien d'autres usages encore.

SYNTAXE

```
document.viewport.getDimensions() -> { width: nombre, height: nombre }
document.viewport.getWidth() -> nombre
document.viewport.getHeight() -> nombre
document.viewport.getScrollOffsets -> offset
```

On retrouve exactement la même forme d'API que pour `Element` (cette cohérence omniprésente fait partie des grandes forces de Prototype, à mon avis). `getDimensions` vous donne la taille du `viewport`, que vous pouvez récupérer de façon individuelle avec `getWidth` et `getHeight`. Attention aux performances cependant : si vous avez besoin des deux, comme pour `Element`, préférez un appel unique à `getDimensions`, qui fait tout le travail et est utilisé en interne par les deux autres.

Quant à `getScrollOffset`, il indique la quantité de défilement en vigueur au niveau du document, dans le même format que pour les méthodes `...Offset` d'`Element` : un tableau à deux éléments dupliquant ses informations dans ses propriétés `left` et `top`.

Selector, l'objet classieux

Voici l'objet utilisé, en interne, par la fonction globale `$$`. Il permet de représenter une série contiguë de sélecteurs CSS (série que j'appellerai par la suite *sélection*). Les sélecteurs pris en charge représentent la quasi totalité des sélecteurs CSS 3, indépendamment du niveau de prise en charge par votre navigateur, plus quelques sélections complémentaires :

- Élément (nom de la balise) ;
- identifiant (`#id`) ;
- Attribut :
 - Présence : `[attr]` ;
 - Valeur : `[attr="value"]` ;
 - Non-valeur : `[attr!="value"]` ;
 - Segment de valeur (basé sur espaces) : `[attr~="value"]` ;
 - Segment de valeur (basé sur tirets) : `[attr|="value"]` ;
 - Partie de valeur (n'importe où) : `[attr*="value"]` ;
 - Début de valeur : `attr[^="value"]` ;
 - Fin de valeur : `attr[$="value"]` ;
- Classe (`.laClasse`) ;
- Descendant (espace, comme dans `div p`) ;
- Fils (`#mainList > li`) ;
- Élément frère suivant (`h2 + p`) ;
- Élément frère ultérieur (`h2 ~ p`) ;
- Pseudo-classes :
 - `:first-child, :first-of-type, :last-child, :last-of-type` ;
 - `:only-child, :only-of-type` ;
 - `:nth-child, :nth-of-type, :nth-last-child, :nth-last-of-type` (avec les syntaxes `even`, `odd`, `a` et `a+b`) ;
 - `:empty, :checked, :disabled, :enabled` ;
 - `:not (yes!)`.

En somme, c'est l'orgie. Pour ceux d'entre vous qui n'ont pas l'habitude de CSS 3 et osent à peine utiliser CSS 2 parce qu'ils doivent prendre en charge MSIE 6 ou 7, pouvoir extraire des éléments du DOM sur la base de CSS 3 deviendra vite addictif...

Puisque `Selector` gère tout cela, `$$` aussi. En fait, `$$` se résume à l'appel approprié d'une méthode statique de `Selector`.

Un mot sur les performances : `Selector` est utilisé de façon critique un peu partout dans Prototype, et `$$` est la deuxième fonction la plus populaire chez les utilisateurs. Il n'est donc pas étonnant que `Selector` fasse partie des modules dont l'optimisation est permanente. Aujourd'hui, une saine compétition existe entre les principales bibliothèques JavaScript pour fournir un service d'extraction DOM basé sur CSS qui soit le plus performant possible.

Lorsque la première édition de ce livre est sortie, utiliser `$$` et les services dérivés n'était pas sans conséquence en termes de performance : dans la mesure du possible, on conseillait par exemple d'utiliser plutôt `getElementsByClassName` lorsque cela suffisait. Mais avec la version 1.5.1, tout ceci a changé. `Selector` a été intégralement réécrit par Andrew Dupont et votre serviteur, pour prendre en charge beaucoup plus de sélections tout en optimisant le code au maximum ; c'est notamment depuis cette version que, lorsque le navigateur prend en charge DOM niveau 3 XPath, `Selector` s'en sert (c'est infiniment plus rapide qu'une extraction en pur JavaScript).

Andrew a assuré l'essentiel du travail sur `Selector` depuis un bon moment, et on lui doit notamment la possibilité récente de tirer parti d'une prise en charge de l'API W3C Selector Query, qui standardise les fonctions que remplit `Selector`. Là aussi, quand c'est présent (Safari 3.1 notamment, avec Firefox et Opera qui travaillent sur la question), ça va très, très vite.

En bref, `Selector` et `$$` vont, aujourd'hui, très vite.

Le seul intérêt de manipuler `Selector` directement, plutôt que de passer par `$$`, consiste à vouloir « cacher » une analyse de règle CSS (sans espaces) pour une utilisation massive.

SYNTAXE

```
Selector.findElement(elements, expression, index) -> eltÉtendu
Selector.findChildElements(rootEl, expression) -> [eltÉtendu...]
Selector.matchElements(elements, expression) -> [eltÉtendu...]
new Selector(expr)
sel.findElements([scope = document]) -> [eltÉtendu...]
sel.match(element) -> booléen
sel.toString() -> chaîne
sel.inspect() -> chaîne
```

Comme vous pouvez le voir, `Selector` s'utilise de deux façons : soit au travers de ses méthodes statiques (`findElement`, `findChildElements` et `matchElements`), soit en construisant une instance de `Selector` pour exploiter ensuite ses méthodes d'instance. Ce deuxième mode n'a d'intérêt que si vous souhaitez réutiliser le sélecteur ainsi « pré-compilé ».

L'API statique possède deux méthodes qui filtrent une liste d'éléments sur la base d'une sélection CSS : `findElement` recherche la *n*ème correspondance parmi les éléments passés (*n* valant par défaut zéro pour la première correspondance), tandis que `matchElements` renvoie tous les éléments correspondants, dans l'ordre initial de la liste.

La méthode statique `findChildElements` accepte un élément racine et recherche tous les descendants correspondant à la sélection. C'est elle qui est utilisée en interne par `$$` :

```
function $$() {
  return Selector.findChildElements(document, $A(arguments));
}
```

Lorsqu'on a un sélecteur précompilé (une instance de `Selector`), on dispose de plusieurs méthodes pour l'exploiter. `findElements` joue un rôle similaire au `findChildElements` statique : on fournit un élément racine, et on filtre les descendants sur la base de la sélection courante. Plus ciblé, `match` détermine simplement si l'élément passé satisfait la sélection ; c'est cette méthode qui rend `Selector` compatible avec la méthode `grep` de `Enumerable`...

Enfin, on a le traditionnel `inspect` (qui renvoie une chaîne au format '#<Selector:expressionPasséeAuConstructeur>') et `toString`, qui renvoie assez logiquement juste l'expression CSS passée à la construction...

Manipulation de formulaires

Prototype fournit de nombreux objets dédiés à la manipulation des formulaires et de leurs champs. On trouve d'abord `Field` et `Form`, qui travaillent respectivement au niveau du champ et du formulaire entier.

Certaines classes, dites *observateurs*, réagissent aux événements de modification. `Form.Observer` réagit à toute modification dans un champ quelconque du formulaire ; il repose en fait sur une série d'objets `Field.Observer`.

Field / Form.Element

L'objet `Form.Element` (et son alias `Field`, que je vais lui préférer dans la suite de cette section) fournit des méthodes de manipulation d'un ou plusieurs champ(s) de formulaire. On fournit comme d'habitude soit l'identifiant, soit une référence directe à l'élément.

J'insiste : l'identifiant, pas le nom de champ (attribut `id`, pas `name`) !

Ces méthodes sont accessibles de façon procédurale (e.g. `Field.activate(champ)`) et directe sur des champs étendus, comme d'habitude (e.g. `champ.activate()`). Mais comme deux d'entre elles ne sont pas disponibles en mode étendu – car déjà présentes en natif – j'ai utilisé le mode étendu quand c'était possible dans le listing des méthodes que voici, pour mieux différencier les deux catégories.

SYNTAXE

```
champ.activate() -> champEtendu
champ.clear() -> champEtendu
champ.disable() -> champEtendu
champ.enable() -> champEtendu
Field.focus(champ) -> champEtendu
champ.getValue() -> value | [value...]
champ.present() -> booléen
Field.select(champ) -> champEtendu
champ.serialize() -> chaîneURLEncodée
champ.setValue(value) -> champEtendu
```

Les noms parlent d'eux-mêmes, à quelques précisions près.

Commençons par les méthodes sans surprise : `clear` efface le champ passé, `focus` donne... le focus (fait du champ la cible des saisies clavier), et `select` sélectionne le texte à l'intérieur du champ (utile uniquement pour les saisies de texte : type `text`, type `password` et balise `textarea`).

En revanche, `focus` et `select` existent en natif sur les champs (`select` uniquement pour le type `text` et la balise `textarea`). Pour ne pas les « écraser », Prototype les met à disposition en syntaxe procédurale. La différence ? La version Prototype renvoie le champ étendu, permettant le chaînage d'appels.

```
// Ne fonctionnera pas ! Le focus() natif ne renvoie pas le champ
$('#edtName').focus().clear();
// Les deux fonctionnent :
$('#edtName').clear().focus();
Field.focus('edtName').clear();
```

La méthode `present` renvoie `true` uniquement si le champ passé a une valeur non vide (au sens strict : si elle ne contient que du *whitespace*, elle sera tout de même considérée remplie ; pour être plus flexible et ignorer le *whitespace*, on peut faire un `getValue().blank()`).

La méthode `activate` est une sorte de *combo* : elle appelle d'abord `focus`, et si le type de champ s'y prête, elle appelle ensuite `select`. Globalement préférable à `focus`, donc.

La méthode `serialize` fournit une représentation URL-encodée du champ, avec son nom et sa valeur. Si le champ est à valeur multiple (cas d'un `select` en mode `multiple`), il est présent autant de fois que nécessaire. La sérialisation respecte à la lettre l'algorithme décrit dans la recommandation W3C de HTML 4.01 pour les formulaires (pour les champs à sérialiser, le format de leur valeur, etc).

La méthode `getValue` renvoie la valeur du champ. Pour un champ à valeur simple, celle-ci est directement renvoyée. En valeur multiple, on obtient un tableau des valeurs sélectionnées.

Vous vous souvenez de la fonction globale `$F` ? C'est en réalité un alias de `Field.getValue` !

On a la méthode réciproque, `setValue`, qui prend une valeur de type approprié (unique ou tableau) et « cale » le champ sur cette valeur. C'est bien pratique pour remplir dynamiquement un formulaire à partir de son état sérialisé, par exemple.

Ces deux méthodes reposent en fait lourdement sur l'objet technique interne `Field.Serializers`.

L'objet `Field` est aussi utilisé en interne par l'objet `Form`, qui utilise une sorte de composition/délégation pour de nombreux traitements.

Un mot sur `Field.Serializers`

Cet objet technique contient des méthodes de « routage » vers le bon traitement d'extraction de valeur. Ses méthodes renvoient en réalité un tableau à deux éléments : le nom du champ et sa valeur.

Je ne rentrerai pas dans les détails, mais voici l'essentiel de son comportement qui, encore une fois, suit à la lettre les règles édictées par le W3C :

- Pour un champ `input` de type `submit`, `hidden`, `password`, `search` ou `text`, ainsi que pour un champ `textarea`, la valeur est utilisée telle quelle, dans tous les cas.
- Pour un champ `input` de type `checkbox` ou `radio`, on ne renvoie quelque chose que si le champ est coché (et inversement, on ne coche que si la valeur n'est pas équivalente à `false`).

- Pour un champ `select` simple (pas d'attribut `multiple="multiple"`), on récupère l'option sélectionnée. S'il n'y en a pas, on renvoie `null`. Si l'option sélectionnée n'a pas d'attribut `value`, son texte est utilisé à la place.
- Pour un champ `select` multiple, la valeur est un tableau des valeurs sélectionnées, chacune étant déterminée comme ci-dessus (attribut `value` si présent, texte sinon).
- Pour un champ `button`, la valeur est le fragment HTML à l'intérieur de l'élément.

Form

L'objet `Form` permet de manipuler un formulaire dans sa globalité. L'argument `form` est bien sûr soit l'identifiant, soit la référence directe de l'élément `form`. Là encore, l'une des méthodes n'est accessible qu'en mode « procédural » et j'ai donc signalé cette différence dans le bloc de syntaxe.

SYNTAXE

```
formulaire.disable() -> formulaireEtendu
formulaire.enable() -> formulaireEtendu
formulaire.findFirstElement() -> eltEtendu
formulaire.focusFirstElement() -> formulaireEtendu
formulaire.getElements() -> [eltEtendu...]
formulaire.getInputs([typeName][, name]) -> [eltEtendu...]
formulaire.request([options]) -> formulaireEtendu
Form.reset(form) -> formulaireEtendu
formulaire.serialize([options]) -> chaîneURLEncodée
formulaire.serializeElements(elements[, options]) -> chaîneURLEncodée
```

Commençons par les méthodes simples : `disable` et `enable` désactivent et ré-activent l'ensemble des champs du formulaire. Un champ qui avait le focus le perdra juste avant de se désactiver. La liste des champs est obtenue par `getElements`. Pratique si l'envoi manuel du formulaire donne un traitement Ajax pendant lequel un second envoi serait problématique.

La méthode `reset()` se contente de réinitialiser le formulaire, comme le bouton obtenu par `<input type="reset" />`. Je rappelle qu'il s'agit simplement de ramener les champs à la valeur qui leur est donnée dans le HTML. Comme elle existe déjà en natif, elle n'est disponible ici qu'en mode procédural, et en profite pour renvoyer le formulaire étendu.

La méthode `getElements` renvoie un tableau de tous les éléments constituant les champs du formulaire, dans l'ordre du document (contrairement à la version 1.5.0).

La méthode `getInputs` est plus spécifique. Elle ne s'occupe que des éléments `input`, et permet de filtrer le résultat sur la base du type ou du nom (ou les deux). Imaginons

que vous ayez des boutons radio : ceux représentant les variantes d'une même donnée auront le même nom de champ. Vous pouvez les obtenir comme ceci :

```
|| $('mainForm').getInputs(null, 'newsletterMode')
```

Ou même plus spécifiquement, pour plus de sécurité :

```
|| $('mainForm').getInputs('radio', 'newsletterMode')
```

À moins que seules les cases à cocher, quelles qu'elles soient, vous intéressent :

```
|| $('mainForm').getInputs('checkbox')
```

La méthode `findFirstElement` renvoie le tout premier élément visible et actif (c'est-à-dire qui ne soit ni `disabled`, ni un `input` de type `hidden`) du formulaire. Toutefois, si au moins l'un des champs ainsi filtrés possède un attribut `tabindex`, celui ayant le `tabindex` de plus petite valeur sera préféré, ce qui est conforme à l'intention du développeur web.

La méthode `focusFirstElement` est une combinaison de confort : elle appelle `activate` sur le résultat de `findFirstElement`, tout simplement.

Nous verrons la méthode `request` au chapitre 7, car elle s'occupe spécifiquement d'envoyer le formulaire en Ajax.

La méthode `serializeElements` crée une représentation sérialisée (URL-encodée, spécifiquement) pour un ou plusieurs élément(s) du formulaire, conformément aux règles édictées par la recommandation W3C de HTML 4.01 pour l'envoi des formulaires. On peut toutefois éviter la forme textuelle URL-encodée pour ne récupérer que les valeurs individuelles pré-sérialisées sous la forme d'un objet anonyme, en passant une option `hash` à `true`.

Pour la petite histoire, si vous voulez simuler un envoi de formulaire par `<input type="image" />` (qui envoie normalement deux paramètres `x` et `y` correspondant à la position du clic sur l'image), vous pouvez passer les options `x` et `y` correspondantes... En pratique, c'est plutôt rare !

Enfin, la méthode `serialize` renvoie la représentation URL-encodée du formulaire, du genre de ce qu'on peut avoir besoin de transmettre comme paramètres GET ou comme corps POST dans une requête Ajax. Elle appelle en fait `serializeElements` en lui passant tous les éléments du formulaire, tels que renvoyés par `getElements`.

Soit le formulaire suivant :

```
<form id="frmMain" method="get" action="/search">
  <p><input type="text" name="q" value="jaubert" /></p>
  <p>
    <input type="checkbox" name="case" value="yes" id="chkCase"
      checked="checked" />
    <label for="chkCase">Respecter la casse</label>
  </p>
  <p><input type="image" alt="Chercher" name="go"
    src="/i/search.gif" /></p>
</form>
```

Voici quelques exemples rapides d'appels :

```
$('#frmMain').serializeElements(['edtSearch'])
// => 'q=jaubert'
$('#frmMain').serializeElements(['edtSearch', 'chkCase', { hash: true }])
// => { q: 'jaubert', 'case': 'yes' }
$('#frmMain').serialize()
// => 'q=jaubert&case=yes&go.x=0&go.y=0'
$('#frmMain').serialize({ x: 42, y: 21 })
// => 'q=jaubert&case=yes&go.x=42&go.y=21'
```

Form.Observer

Voici notre premier exemple d'*observateur*. Dans Prototype, un observateur est un mécanisme surveillant à intervalles réguliers la valeur de quelque chose. Lorsque celle-ci change (ainsi qu'à la première observation, lors de laquelle on n'a pas encore de valeur précédente à comparer), l'observateur appelle une fonction de rappel qui lui aura été fournie à la construction.

SYNTAXE

```
new Form.Observer(form, intervalInSecs, callback)
```

Techniquement, les observateurs sont définis par l'objet `Abstract.TimedObserver`, qui est étendu par divers objets, dont `Form.Observer`. Ces objets ont juste besoin d'implémenter une méthode `getValue()`, qui renvoie la « valeur » ainsi observée.

Un observateur a toujours trois arguments dans son constructeur :

- 1** L'élément observé (ici un formulaire).
- 2** L'intervalle (ou la période, pour les mordus) en secondes.

3 La fonction de rappel. Celle-ci recevra deux arguments : l'élément observé et sa nouvelle « valeur ».

Afin de prendre en compte l'ensemble de ses champs, un `Form.Observer` définit le résultat de `Form.serialize` comme étant sa « valeur ».

Voici un exemple d'utilisation (qui semble bien pénible pour l'internaute) :

```
function formChanged(form, newValue) {
    alert($H(newValue.parseQuery()).toArray().join('\n'));
}
new Form.Observer('mainForm', 1, formChanged);
```

On peut imaginer utiliser ce genre de choses pour soumettre les modifications à la volée en Ajax, par exemple. Si c'est pour de la complétion automatique de texte, je vous conseille toutefois plutôt d'utiliser l'objet `Ajax.AutoComplete` de script.aculo.us. Une merveille, et déjà tout prêt !

Field.Observer

Cet objet permet de réagir à intervalles réguliers au changement d'un seul champ plutôt que de n'importe quel champ du formulaire.

SYNTAXE

```
new Field.Observer(elt, intervalInSecs, callback)
```

La « valeur » est bien sûr obtenue par `Field.getValue...`. Cela s'utilise comme les autres observateurs (voir l'exemple pour `Form.Observer`).

Gestion unifiée des événements

Je l'ai signalé à de multiples reprises au chapitre précédent, Prototype brille tout particulièrement par sa capacité à unifier la gestion des événements. On l'a vu, tous les navigateurs répandus ne se conforment pas toujours exactement au DOM niveau 2. Événements, et MSIE est particulièrement à côté de la plaque en adoptant une approche entièrement distincte du standard. Cette situation constitue souvent un casse-tête pour les développeurs web souhaitant mettre en œuvre des traitements événementiels un tant soit peu étoffés.

Au travers de son objet `Event`, Prototype nous fournit tout le nécessaire pour associer ou révoquer des gestionnaires d'événements, et analyser les détails d'un événement lorsqu'il survient.

La version 1.6 s'accompagne d'une refonte totale de l'API d'événements ; si vous l'avez découverte dans la première édition de ce livre, je ne saurais trop vous encourager à la relire ici attentivement, tant les changements sont nombreux et, parfois, très importants.

Event

L'objet Event est normalement fourni par le DOM, et Prototype le « garantit ». Il sert par ailleurs d'espace de noms pour les méthodes liées à la gestion événementielle, méthodes qui sont désormais toutes disponibles directement sur les objets événements reçus par vos gestionnaires, sur MSIE comme ailleurs.

Cet objet est largement étendu et augmenté par Prototype de façon à fournir une API unifiée, à garantir un certain nombre de propriétés, etc. Tout ceci permet de n'écrire *qu'un* code qui passe partout, ce qui simplifie grandement notre travail.

Observer (ou cesser d'observer) un événement

Lorsqu'on commence à s'intéresser à un événement, on dit qu'on *l'observe*. On définit une fonction gestionnaire de l'événement, calée sur un élément du DOM (le plus haut niveau étant l'objet *window*). On peut plus tard cesser cette observation, lorsque l'événement ne nous intéresse plus.

Le DOM prévoit deux méthodes de propagation des événements : d'abord en descente, de la fenêtre vers l'élément source, à des fins de censure. On appelle ce mode la *capture*, mais MSIE ne le prend pas en charge et pour cette raison, depuis la version 1.6, Prototype l'a retiré de son API car il ne peut unifier cette possibilité. Faute de censure par capture, la plupart des événements vont *bouillonner*, c'est-à-dire se déclencher sur l'élément source puis chaque élément conteneur en remontant la chaîne des éléments parents dans le DOM. Cette propagation peut également être interrompue à tout moment par un appel explicite, mais c'est elle qui nous fait dire qu'on observe tout un fragment lorsqu'on observe l'élément racine de ce fragment.

Prototype permet donc de limiter notre observation à un fragment du DOM précis (on observant au niveau de l'élément racine de ce fragment), ou d'observer tout le document (objet *document*), voire d'observer la fenêtre (objet *window*), par exemple pour son événement *load*. Suivant le cas, on aura le choix entre trois syntaxes :

SYNTAXE

```
elt.observe(nomÉvénement, gestionnaire) -> eltÉtendu  
document.observe(nomÉvénement, gestionnaire) -> document  
Event.observe(objet, nomÉvénement, gestionnaire) -> objet
```

Lorsqu'on a sous la main un élément DOM étendu, on peut utiliser sa méthode `observe` pour s'intéresser à l'événement dans le cadre du fragment de document dont cet élément est la racine.

Si l'on souhaite observer un événement sur tout le document, ou un élément personnalisé spécifique au document, on utilise `document.observe`, car `document` n'est pas un élément DOM comme les autres.

Enfin, si l'on veut observer l'événement sur un élément DOM qui n'est pas encore étendu ou sur un autre objet, typiquement `window`, on utilise la syntaxe « procédurale » `Event.observe`, en passant l'objet en premier argument.

Attention : le nom de l'événement est son nom DOM (pas de préfixe 'on', donc par exemple 'load' et 'click' et non pas ' onload' et ' onclick').

Pour arrêter d'observer l'événement, on utilise une syntaxe symétrique :

SYNTAXE

```
elt.stopObserving([nomÉvénement[, gestionnaire]]) -> eltÉtendu  
document.stopObserving([nomÉvénement[, gestionnaire]]) -> document  
Event.stopObserving(objet[, nomÉvénement[, gestionnaire]]) -> objet
```

Notez que depuis Prototype 1.6, on ne précise plus obligatoirement à `stopObserving` le gestionnaire inscrit ou même l'événement concerné : afin de faciliter le « nettoyage » des observateurs, on peut ne préciser que le nom de l'événement (ce qui nettoie tous les gestionnaires pour cet événement sur cet élément ou objet), voire rien du tout (ce qui nettoie tous les gestionnaires pour tous les événements sur cet objet).

Voici un premier exemple avec une fonction classique, qui cache un élément dès qu'on clique dessus :

Listing 4.10 Inscription d'une fonction comme gestionnaire d'événement

```
function hideElement(event) {  
    event.element().hide();  
}  
  
$('grandTimide').observe('click', hideElement);
```

Nous verrons la méthode `element()` dans quelques instants.

Voici à présent une autre version, qui garde un compteur de clics pour chaque élément cliqué (ce qui permet d'utiliser le même gestionnaire pour de nombreux éléments, en supposant toutefois qu'ils ont tous des identifiants), et le fait disparaître au bout de 3 clics.

Comme le gestionnaire est une méthode qui a besoin de pouvoir accéder aux champs de son objet, on utilise bind :

Listing 4.11 Inscription d'une méthode comme gestionnaire d'événement

```
var DelayedHider = {
  _countKeeper: {},
  handleClick: function(event) {
    var elt = event.element();
    var count = this._countKeeper[elt] || 0;
    this._countKeeper[elt] = ++count;
    if (3 == count)
      elt.hide();
  }
};
$('petitTimide', 'click',
  DelayedHider.handleClick.bind(DelayedHider));
```

C'est l'occasion de retrouver notre bon vieux || pour gérer les valeurs par défaut, ou comme ici le cas initial, lorsque le compteur n'existe pas dans le tableau associatif représenté par un objet basique.

Que se passe-t-il si l'on n'utilise pas bind ?

Depuis Prototype 1.6, les gestionnaires d'événement sont, par défaut, exécutés dans le contexte de l'élément sur lequel vous avez appelé observe. C'est extrêmement pratique : cela veut dire que dans un tel gestionnaire, this référence forcément le sujet de l'appel à observe, sans que vous ayez besoin de maintenir une fermeture lexicale ou un *binding* pour cela.

Examiner un événement déclenché

Voyons à présent quelles manipulations nous pouvons effectuer sur l'objet événement forcément passé en premier argument à nos gestionnaires.

Coincer le coupable

Lorsqu'un événement survient, votre gestionnaire se réveille. Mais l'élément qui a reçu l'événement n'est pas forcément celui sur lequel votre gestionnaire est inscrit.

SYNTAXE

```
evt.element() -> eltÉtendu
evt.findElement(selecteur) -> eltÉtendu
evt.relatedTarget -> eltÉtendu
```

En vertu du principe de bouillonnement, il peut s'agir d'un élément descendant : ainsi, un gestionnaire `click` pour un paragraphe peut se déclencher si l'on clique sur un texte dans un élément `strong` à l'intérieur de ce paragraphe...

Par ailleurs, vous pouvez parfaitement avoir utilisé un même gestionnaire pour de multiples éléments. Alors comment distinguer ? À l'aide de la méthode `element`.

Cette méthode prend l'objet événement reçu et se débrouille pour extraire une référence à l'élément cible étendu (en 1.5, les méthodes de `Event` ne renvoient pas les éléments étendus, mais 1.6 a corrigé ce défaut).

Il existe également une méthode encore plus utile, `findElement`. Elle permet de récupérer le plus proche élément ancêtre de l'élément « cible » pour une sélection CSS donnée.

Si vous avez observé un élément précis et souhaitez simplement récupérer cet élément dans le gestionnaire, cela ne vous sert guère : `this` référence déjà le sujet de votre appel à `observe`. Donc même si vous avez observé les clics sur un lien et que l'utilisateur clique sur un `` dans le `<a>`, certes, `element` renverra le ``, mais `this` référencera fidèlement votre `<a>`.

Non, là où `findElement` est très utile, c'est en combinaison avec des principes bénéfiques et désormais incontournables comme la *délégation d'événement*.

Prenons l'exemple suivant : vous avez non pas un lien, mais une batterie de liens, au comportement parfaitement similaire (e.g. liste d'éléments de même nature, par exemple des articles sur un blog, chaque article étant doté d'un lien de suppression que vous souhaitez « ajaxifier »).

Plutôt que d'associer un gestionnaire à chaque lien, ou même juste d'observer individuellement chaque lien avec le même gestionnaire, vous décidez de *déléguer* grâce au bouillonnement : vous observez tous les clics dans la liste, et vérifierez le moment venu s'ils ont bien eu lieu sur le lien en question !

Imaginons le fragment suivant :

```
<ul id="articles">
  <li>
    <h2>Il fait chaud...</h2>
    <a href="/admin/articles/1" class="delete">Supprimer</a>
  </li>
  <li>
    <h2>Il fait beau...</h2>
    <a href="/admin/articles/2" class="delete">Supprimer</a>
  </li>
```

```
<li>
  <h2>Et on n'veut plus travailler !</h2>
  <a href="/admin/articles/3" class="deleter">Supprimer</a>
</li>
</ul>
```

Plutôt que d'associer un gestionnaire à, ou simplement d'observer, chaque lien, autant déléguer dans la joie ! Voyez plutôt :

```
function handleArticleDeleters(event) {
  var deleter = event.findElement('a.deleter');
  if (!deleter) return;
  event.stop();
  new Ajax.Request(deleter.href, { method: 'delete',
    onSuccess: function() { deleter.up('li').fade() } });
}
$('articles').observe('click', handleArticleDeleters);
```

On commence par voir si le clic s'est produit au sein d'un lien : c'est là que `findElement` est pratique, puisqu'il nous permet de faire abstraction du balisage au sein du lien : le jour où l'on ajoutera des `span` et Tim Bernes-Lee sait quoi d'autre, `findElement` s'y retrouvera quand même ! On ne peut pas utiliser `this`, puisqu'il référence le sujet d'`observe`, donc la liste tout entière...

Ensuite, si l'on n'a pas trouvé un tel lien dans nos parents, et donc que le clic n'a pas eu lieu à un endroit qui nous concerne, on fait avorter le gestionnaire (`return`). En revanche, si nous sommes concernés, on commence par abandonner le traitement normal du clic (naviguer vers la cible du lien) et arrêter la propagation ; c'est le rôle de `stop`, sur lequel nous reviendrons un peu plus loin.

Il ne nous reste plus qu'à effectuer une requête Ajax appropriée (on suppose ici que le serveur est conforme aux principes REST et attend une requête HTTP DELETE sur l'URL identitaire de l'article), qui si elle est validée fera disparaître en fondu l'élément de liste concerné.

Tout beau, tout propre.

Si aucun élément ancêtre ne correspond à la sélection, `findElement` renvoie `null`.

Prototype 1.6 garantit aussi désormais la propriété `relatedTarget`, qui ne sert qu'aux événements de souris `mouseover` (entrée sur un élément) et `mouseout` (sortie d'un élément). Elle indique alors l'élément « d'en face » : quand vous entrez sur un élément (`mouseover`), c'est celui dont vous venez, du coup, de sortir. Quand vous sortez d'un élément (`mouseout`), `relatedTarget` indique celui sur lequel vous venez d'entrer. C'est une propriété bien utile lorsqu'on souhaite implémenter de façon portable

certains événements comme `mouseenter` et `mouseleave`, souvent bien plus utiles que `mouseover` et `mouseout`.

Étouffer l'affaire

La plupart du temps, vos gestionnaires sont les seuls censés devoir traiter un événement particulier (par exemple un clic spécifique ou l'envoi d'un formulaire). Dans de tels cas, vous pouvez vouloir stopper la propagation de l'événement.

Mais peut-être votre gestionnaire implémente-t-il un contrôle validant le comportement par défaut de l'événement (on pense principalement aux envois de formulaire). Si votre algorithme le réclame, il faut pouvoir annuler ce comportement par défaut.

SYNTAXE

```
evt.preventDefault()  
evt.stop()  
evt.stopPropagation()
```

Ces deux opérations étaient auparavant indissociables avec Prototype : on considérait que si vous annuliez le comportement par défaut d'un événement, le propager eût risqué d'induire d'autres gestionnaires en erreur, tandis que si vous annuliez la propagation, c'était probablement que le comportement par défaut ne vous intéressait pas.

En pratique, c'est généralement vrai. La méthode utilisée alors est `stop`. Elle interrompt la propagation de l'événement tout en annulant son comportement par défaut. C'est très pratique.

Toutefois, il peut arriver que vous ayez besoin d'interrompre la propagation sans pour autant annuler le comportement par défaut, ou inversement. Prototype 1.6 garantit donc désormais les deux méthodes du DOM niveau 2 Événements correspondantes : `stopPropagation` et `preventDefault`, que MSIE n'implémente pas encore telles quelles.

Micro-détail : un événement sur lequel on a appelé `stop` le signale en arborant alors une propriété `stopped` de valeur `true`. Cette dernière peut se révéler utile lorsqu'on implémente des événements personnalisés un peu complexes comme la prise en charge portable des défilements par la molette de la souris.

Déterminer l'arme du crime

Savoir que l'événement s'est produit, et sur quel élément il a originellement eu lieu, ne suffit pas toujours. Loin de là ! Lorsque tous les navigateurs implémentent de

façon uniforme une information, Prototype ne rajoute rien. En revanche, lorsqu'on a des disparités, il fournit une API unifiée.

L'état de la souris

Il nous est possible d'examiner l'état des boutons et la position du curseur dans plusieurs référentiels de coordonnées.

SYNTAXE

```
evt.isLeftClick() -> booléen  
evt.isMiddleClick() -> booléen  
evt.isRightClick() -> booléen  
evt.pageX -> Number  
evt.pageY -> Number  
evt.pointer() -> Number  
evt.pointerX() -> Number  
evt.pointerY() -> Number
```

Les boutons de la souris sont représentés de façon assez diversifiée d'un navigateur à l'autre, sans parler de la plate-forme (n'oubliez pas que sur Mac, par exemple, on n'a pas de clic droit : on utilise généralement *Ctrl+Clic*). Qui plus est, certains navigateurs utilisent une propriété *button*, d'autre une propriété *which*, et les valeurs vont soit de 1 à 3, soit de 0 à 2.

Prototype uniformise la détection de tout ceci au travers de trois méthodes : *isLeftClick*, *isMiddleClick* et *isRightClick*.

La position du curseur est aussi un concept à géométrie variable. La grande question est en effet : dans quel référentiel ? La page ? La portion visible de la page ? L'écran ?

Le DOM niveau 2 Événements prévoit deux couples de propriétés pour l'objet événement que vous recevez :

- *screenX* et *screenY* : position à l'écran ;
- *clientX* et *clientY* : position dans la partie visible de la page.

Prototype prévoit par ailleurs les besoins de positionnement et de glisser-déplacer en définissant la notion de position *dans le document complet*, au travers des deux méthodes *pointerX* et *pointerY*, ou carrément de *pointer* tout court (qui renvoie un objet basique de position comme on en a déjà vu : une propriété *x* et une propriété *y*).

Prototype va même jusqu'à garantir les propriétés équivalentes *pageX* et *pageY* sur l'objet événement, ce qui est encore plus simple et rapide d'utilisation !

Les touches du clavier

Côté clavier, sachez d'abord que Prototype définit une série de constantes pour la propriété `keyCode` des événements, afin de s'y retrouver un peu plus facilement :

```
Event.KEY_(BACKSPACE/DELETE/DOWN/END/ESC/HOME/INSERT/LEFT/PAGEUP/  
PAGEDOWN/RETURN/RIGHT/TAB/UP)
```

Les touches correspondant à des caractères ASCII n'ont pas besoin de constantes, car leur valeur correspond au caractère recherché. Si vous préférez, une constante `KEY_8` serait égale au code du caractère '`8`'. Je rappelle que `BackSpace` correspond à la touche française `Ret.Arr.`, `Delete` à notre `Suppr`, `Home` à notre `Orig` et `End` à notre `Fin`.

Sur quoi utiliser ces constantes et valeurs, me direz-vous ? Eh bien, il se trouve que les principaux navigateurs implémentent de façon homogène l'information, pour une fois. Utilisez la propriété `keyCode` de l'objet événement que votre gestionnaire reçoit, comme ceci :

```
function handleKeyPress(event) {  
    if (KEY_RETURN == event.keyCode)  
        // code de traitement  
}
```

Les événements clavier et souris sont par ailleurs toujours dotés de propriétés booléennes indiquant l'état d'enfoncement des principaux modificateurs : `Alt`, `Ctrl` et `Maj`. Ce sont les propriétés `altKey`, `ctrlKey` et `shiftKey`, respectivement. Vous pouvez donc facilement réagir à un `Ctrl+G` :

```
function handleKeyDown(event) {  
    if ('G'.charCodeAt(0) == event.keyCode && event.ctrlKey)  
        // Code de traitement  
}
```

Le `charCodeAt(0)` est nécessaire car '`G`' est une `String`, par un `char` (qui n'existe pas en JavaScript). L'expression '`G`' == 71 donne `false`. Il faut donc obtenir le code numérique du premier caractère de cette `String`.

Pour conclure sur les méthodes étendues de l'objet `Event`, un petit détail : Prototype les dote de l'incontournable `inspect`, qui renvoie simplement '[object Event]'. À tout hasard...

Utiliser les événements personnalisés

Prototype 1.6 introduit la notion d'événements personnalisés, c'est-à-dire des événements DOM dotés des comportements et propriétés habituels (élément cible, bouillonnement, arrêt de propagation, etc.) mais déclenchés par nos soins.

Ces événements sont extrêmement utiles puisqu'ils permettent de :

- Réduire le couplage de code. Par exemple, en associant une fonction locale ou une méthode privée à un événement personnalisé, ce code pourra être déclenché depuis n'importe quel point dans les scripts en déclenchant l'événement en question. Pour ceux qui connaissent, c'est un peu le principe signal/slot de Qt et Objective C.
- Implémenter de façon portable des événements encore « exotiques », qui soit ne sont pas encore normalisés dans le DOM. Événements, soit sont 100 % propriétaires et disponibles de façons diverses dans les navigateurs. Parmi les cas classiques, on trouve les événements de molette de souris (`mousewheel/DOMMouseScroll`) et d'*espace aérien* » (`mouseenter` et `mouseleave`).

Pour être détecté comme personnalisé, notre événement doit avoir un *espace de noms*, à savoir comporter au moins un signe deux-points (:). Par exemple, `mouse:wheel`, `content:removed`, `ajax:failure` ou `dom:loaded`.

Un événement personnalisé se traite comme n'importe quel événement : avec `observe` et `stopObserving`. En revanche, on doit pouvoir le *déclencher*. On retrouve notre API tripartite :

SYNTAXE

```
elt.fire(nomÉvénement[, memo]) -> eltÉtendu  
document.fire(nomÉvénement[, memo]) -> document  
Event.fire(objet, nomÉvénement[, memo]) -> objet
```

Eh oui, pour déclencher un événement personnalisé, il suffit d'appeler `fire` sur l'élément DOM ou l'objet concerné. Il bouillonnera normalement jusqu'à interception éventuelle, etc.

Afin d'offrir un maximum de flexibilité, on peut doter l'objet événement d'une propriété `memo`, qui recevra la valeur du dernier argument, optionnel. Cela permet d'associer à l'objet événement toute information supplémentaire (par exemple, pour un `mouse:wheel`, on pourrait avoir un `memo` avec une propriété `delta` indiquant le nombre de crans du défilement).

L'événement personnalisé dom:loaded de l'objet document

Prototype 1.6.0 fournit un seul événement personnalisé, mais de taille : `dom:loaded`, disponible sur l'objet global `document`. Nous l'avons déjà utilisé tout au long de ce chapitre, et nous continuerons pour le restant du livre.

Contrairement à l'événement `load` de l'objet `window`, qui se déclenche une fois la page intégralement chargée (y compris les centaines de Ko, voire les Mo, d'images, d'animations Flash, etc.), `dom:loaded` se charge une fois *qu'on peut scripter la page*.

Et c'est quand, ça ? Eh bien, jusqu'à Prototype 1.6.0.2 inclus, c'était une fois le DOM chargé (c'est le plus important, après tout), d'où le nom. Prototype 1.6.0.3 semble décaler ça très légèrement, et attend que les feuilles de styles (CSS) soient également chargées (et donc que le navigateur soit sur le point d'effectuer un premier *rendering utile*).

Notez au passage que vous pouvez savoir, depuis n'importe où dans votre code, si l'on a atteint ce stade du chargement : Prototype définit à ce moment-là une propriété `loaded` à `true` sur l'objet global `document`.

Pour terminer, sachez qu'au moment où j'écris ces lignes, Prototype Core prévoit de mettre en place un certain nombre d'événements personnalisés dans la version 1.6.1, notamment autour d'Ajax et des modifications de contenu basées sur `update`, `replace` et `insert`. À suivre !

Form.EventObserver

`Form.EventObserver` est similaire à `Form.Observer` : il s'agit de déterminer si un champ du formulaire a changé de valeur depuis le dernier examen, et le cas échéant, d'appeler une fonction de rappel.

SYNTAXE

```
| new Form.EventObserver(form, callback)
```

Mais au lieu d'examiner périodiquement le formulaire, on réagit aux événements. En l'occurrence, un `Form.EventObserver` enregistre un gestionnaire pour tous les champs du formulaire (événement `click` pour les cases à cocher et boutons radio, événement `change` pour les autres). C'est donc plus immédiat que `Form.Observer`, ce qui n'est pas forcément mieux... Tout dépend de l'ergonomie recherchée.

Attention au piège habituel : `Form.EventObserver` crée en fait des `Field.EventObserver` (voir section suivante) pour chaque champ du formulaire *au moment de l'instanciation*. Si vous ajoutez dynamiquement des champs au formulaire par la suite, leurs modifications ne déclencheront donc pas cet observateur...

Field.EventObserver

On retrouve la relation Form / Field déjà vue pour les observateurs basés sur un intervalle de temps. Il s'agit ici de ne réagir qu'à l'événement de modification d'*un seul* champ.

SYNTAXE

```
| new Field.EventObserver(elt, callback)
```

L'objet global Prototype

Il s'agit plus d'un espace de noms que d'un objet. On y trouve d'abord `Prototype.Version`, une constante indiquant la version exacte de Prototype, par exemple « 1.6.0.3 ». C'est utile pour vérifier une dépendance. La bibliothèque `script.aculo.us`, par exemple, s'en sert pour vérifier qu'on l'utilise avec une version suffisamment récente de Prototype. Ainsi, la version 1.8.1, qui dépend de Prototype 1.6.0, se sert-elle de cette donnée.

Par ailleurs, Prototype renferme deux fonctions simplistes, fort utiles dans de nombreux emplois de méthodes nécessitant un itérateur (une notion que nous avons déjà vue avec `Enumerable`) :

- `Prototype.emptyFunction` est une fonction vide, comme son nom l'indique. Elle ignore ses arguments, ne fait rien, et ne renvoie rien. Prototype s'en sert souvent pour éviter d'avoir à gérer le cas d'une fonction optionnelle qui n'aurait pas été fournie, en basculant sur celle-ci au moyen d'un simple opérateur `||`. Nous verrons une illustration de ce mécanisme au chapitre 9, dans notre exemple autour de Flickr.
- `Prototype.K` est la fonction identité : elle renvoie simplement son premier argument. Elle est utilisée par de nombreuses méthodes à itérateur, lorsqu'un itérateur spécifique est manquant. Dans les prochaines sections, vous verrez de nombreuses méthodes ayant un itérateur optionnel en argument. S'il n'est pas fourni, c'est `Prototype.K` qui est utilisé à sa place.

Enfin, Prototype fournit deux objets internes qui permettent à nos scripts d'utiliser certaines conditions relatives au navigateur courant : soit qu'il s'agisse de l'identifier, soit qu'on souhaite déterminer si une possibilité DOM avancée est disponible.

- `Prototype.Browser` renferme cinq propriétés booléennes nommées d'après le type de navigateur détecté, qui permettent de déterminer la famille du navigateur courant : IE, Opera, WebKit (notamment Safari), Gecko (Mozilla/Firefox) et

MobileSafari (iPhone/iPod Touch). Il faut noter que IE et Opera ne se basent pas uniquement sur l'agent utilisateur (la description textuelle que donne le navigateur de lui-même), mais bien sur les caractéristiques techniques de la prise en charge du DOM par le navigateur : par exemple, même si l'on demandait à Safari ou à Firefox de « se faire passer pour MSIE », `Prototype.Browser.IE` vaudra `false`. Prototype se sert de ces propriétés partout où il doit « jongler » avec les incompatibilités inter-navigateurs pour présenter une API unifiée.

- `Prototype.BrowserFeatures` détecte la prise en charge (ou son absence) de certaines possibilités DOM avancées dont il se sert le cas échéant. Ce sont là des données très techniques à usage principalement interne, et je ne les liste brièvement que par souci d'exhaustivité :
 - XPath pour DOM niveau 3 XPath (utilisé par Selector).
 - `SelectorsAPI` est clair (<http://www.w3.org/TR/selectors-api/>)...
 - `ElementExtensions` pour la possibilité d'étendre directement les prototypes des éléments, au lieu de devoir les étendre chacun à la volée.
 - `SpecificElementExtensions` pour vérifier que l'ensemble des éléments DOM ne partagent pas un prototype unique.

Pour aller plus loin...

Sites

- Le site officiel de Prototype, pour récupérer la dernière version finalisée : <http://prototypejs.org/>
- Le site officiel de script.aculo.us, pour disposer de nombreux effets et outils tout prêts, et voir des utilisations avancées de Prototype : <http://script.aculo.us/>
- Le référentiel officiel des bibliothèques complémentaires tierce-partie, certaines étant déjà célèbres : <http://scripteka.com>.

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages offerts par les outils d'indexation et de recherche de Google. On y trouve de nombreux membres ayant un bon niveau technique. Hélas, il a fini par crouler sous le *spam* faute d'une modération active.

Heureusement, à l'initiative de T.J. Crowder, un groupe de remplacement, au nom d'ailleurs plus explicite, a vu le jour en juillet 2008 et concentre désormais toute l'activité, le groupe historique n'existant plus que pour ses archives : <http://groups.google.com/group/prototype-scriptaculous>.

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse sur le canal IRC dédié à Prototype, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#prototype`.

5

Déboguer vos scripts

JavaScript, c'est comme tout : meilleurs sont les outils, et plus on développe vite. Croyez-le ou non, aujourd'hui encore, la plupart des développeurs web n'utilisent pas d'outil évolué dans leur navigateur pour mettre au point leur JavaScript. Ces mêmes développeurs qui frémissent d'horreur à l'idée de devoir développer en Java, C++ ou C# sans un EDI haut de gamme (et ne parlons même pas de Visual Basic), persistent à n'utiliser, pour travailler avec JavaScript, que l'équivalent d'un silex et d'un bout de bois. L'annexe D et cette section visent à vous élever au-dessus de cet état préhistorique.

Et pourtant, l'univers JavaScript propose les mêmes richesses que ceux des autres langages : éditeurs, complétion automatique, véritables EDI même, frameworks de tests unitaires et fonctionnels, extraction automatique de documentation, et j'en passe.

Il ne s'agit pas ici de passer tout cela en revue, mais de nous pencher plus particulièrement sur la question du débogage. Pour citer un membre du projet Mozilla, « déboguer du JavaScript, c'est un peu faire la chasse aux fantômes ». Mais il ne s'agit pas là d'une fatalité. Des outils peuvent vous aider considérablement, et nous allons en voir quelques-uns ici.

Déboguer d'abord sur Firefox avec Firebug

Je le répète : pour mettre au point la page, il n'y a rien de mieux que Firebug ! Son débogueur JavaScript est rapide, léger et simple d'emploi, sans parler de l'incontournable console qui permet de triturer la page en direct.

Côté JavaScript et DOM, il en va finalement comme pour les CSS :

- 1 On met au point confortablement sur Firefox.
- 2 On attaque ensuite le gros morceau de compatibilité : IE6.
- 3 Après quoi on peut vérifier sur IE7, déjà moins gênant.
- 4 On finit avec les « formalités », qui ne posent généralement pas de problème ; IE8 qui pointe son nez, mais aussi Safari et Opera.

Pour le moment donc, restons dans les abords confortables de Firefox, et examinons Firebug. Changement important depuis sa version 1.2 sur Firefox 3, les fonctions de console, de script et de supervision d'Ajax sont désactivées par défaut ; on les active au cas par cas, en fonction du domaine de la page.

La figure 5-1 illustre l'aspect du panneau Firebug (*Ctrl+Maj+L* ou *Cmd+Maj+L* sur Mac OS X) lorsqu'on est sur un site pour lequel on n'a pas encore activé ces fonctionnalités.

On peut cocher les cases qui nous semblent utiles et cliquer sur le bouton *Enable selected panels for...*, ce qui va *recharger la page*. Par la suite, ces options seront conservées. Quand on débogue un site, on active les trois panneaux :

- *Console* pour pouvoir interagir en direct avec les scripts et le DOM de la page, ce qui est indispensable pour être productif ;
- *Script* pour pouvoir déboguer les scripts ;
- *Réseau* pour garder un œil sur les requêtes Ajax.

Cette activation conditionnelle vise à ne pas ralentir votre navigation en initialisant Firebug pour chaque page consultée, mais vous pouvez néanmoins revenir à l'activation totale qu'on avait sur Firebug 1.0 en cliquant sur les flèches à droite des onglets concernés et en choisissant *Enabled*, comme on peut le voir dans la figure 5-2.

(Au passage, si vous n'avez pas l'onglet *YSlow* dans votre Firebug, ne vous inquiétez pas : il s'agit d'une extension supplémentaire pour mesurer les performances d'une page dans un contexte « site à très fort trafic » et vous proposer les optimisations idoines, qui ne sont pas toujours à la portée du premier venu.)

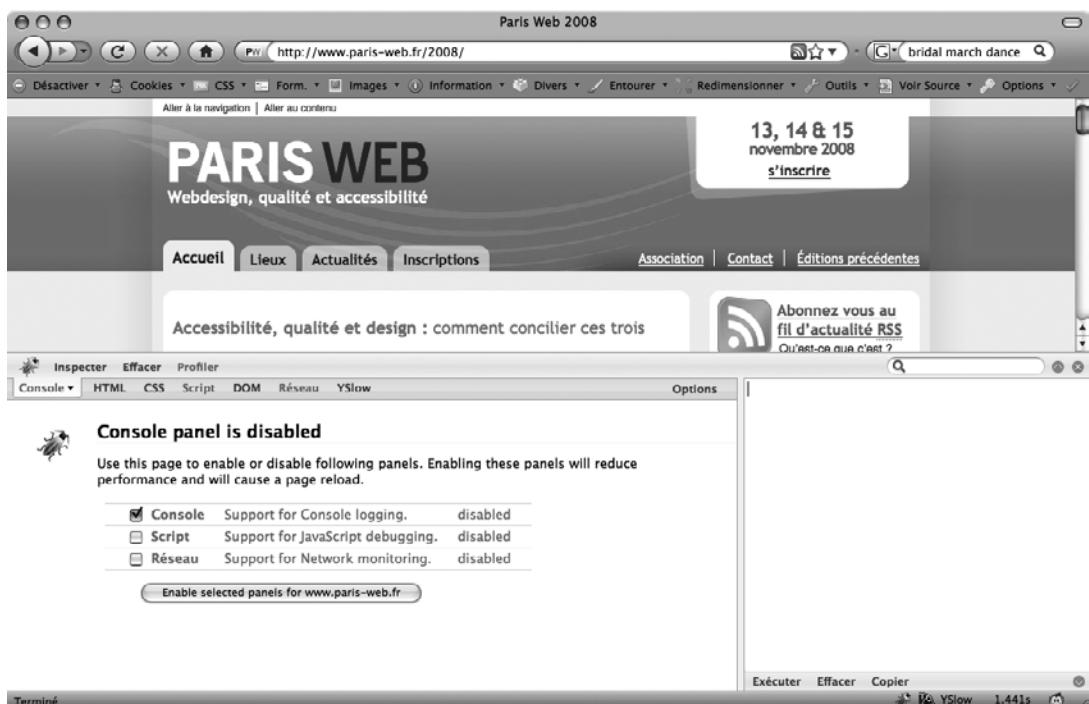
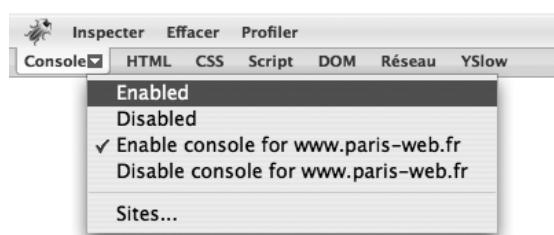


Figure 5–1 Firebug 1.2 en état de base : pas de fonctions script activées

Figure 5–2

Activer une fois pour toutes un panneau de Firebug 1.2



La console : rapide, efficace, et... très sous-utilisée

La console de Firebug est excellente, mais souvent sous-utilisée. Lorsqu'elle est active, c'est le panneau par défaut : il suffit de faire le raccourci (*Cmd+Shift+L* sur Mac OS X, *Ctrl+Shift+L* ailleurs) et hop ! Elle est prête. On peut dès lors y taper n'importe quel morceau de script et l'exécuter directement.

Certaines versions de Firebug ont par défaut une zone de saisie mono-ligne (sous la console elle-même). Il suffit alors de cliquer sur la petite flèche grise à droite de cette zone de saisie pour passer en multi-lignes, sur la droite, ce qui permet de saisir un

fragment complexe ; l'exécution se fait alors par *Ctrl+Entrée* (ou *Cmd+Entrée* sur Mac OS X) au lieu du simple appui sur *Entrée*. Seul inconvénient de cet autre mode : on perd la possibilité de se promener dans l'historique des saisies avec les touches *Haut* et *Bas* ou d'utiliser la complétion sommaire avec *Tab*...

Prenons par exemple le site de Paris Web 2008 : <http://paris-web.fr/2008/>, et ouvrons-y la console. Puis tapons une expression JavaScript simple :

```
| 7 * 6
```

La console affiche bien 42... Corsons un peu la chose :

```
| Math.sin(Math.PI / 2)
```

Et hop, on a bien 1. L'objet Math est un objet natif de JavaScript, auquel nous avons donc accès. Examinons le « DOM niveau 0 » à présent :

```
| location.href
```

Ça nous affiche "<http://www.paris-web.fr/2008/>". Il s'agit d'une chaîne de caractères, et vous remarquez au passage que la console effectue une coloration syntaxique : les nombres en bleu foncé, les chaînes en rouge, etc. Ce `location` est en fait une propriété de l'objet global `window`, qui est le contexte implicite d'exécution de notre code, ce dernier s'exécutant au niveau « racine » des scripts.

Essayons à présent de modifier le titre de la fenêtre :

```
| document.title = "Voilà ce que j'en fais de ton titre !";
```

Miracle : ça marche ! Vous imaginez combien c'est pratique, dès que vous avez des fonctions à tester issues de vos scripts.

La console propose par ailleurs un objet que vous pouvez utiliser dans vos scripts pour déboguer : `console`. Cet objet est infiniment plus pratique que le bon vieux `alert`, car `console` n'interrompt pas l'exécution de votre script en affichant une fenêtre, ce qui pourrait gêner vos tentatives de débogage d'Ajax, de glisser-déplacer, de survol, etc.

L'objet `console` fournit plusieurs méthodes pour envoyer un message vers la console, justement : `log`, `info`, `debug`, `warn` et `error`. En fait, `log` et `debug` s'affichent exactement de la même façon, on choisit donc le nom qu'on préfère, mais les trois autres ont leur couleur et leur icône propres. Essayez donc :

```
console.log('un log');
console.info('une info');
console.debug('un débogage');
console.warn('un avertissement');
console.error('une erreur');
```

La figure 5-3 récapitule nos essais jusqu'à présent :

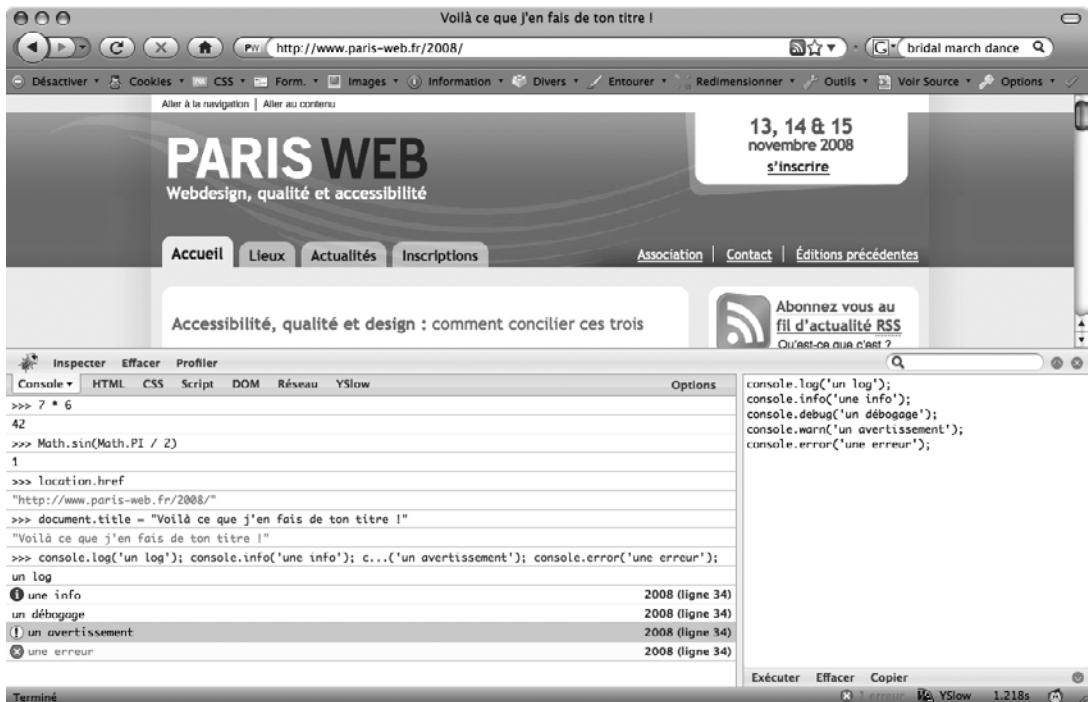


Figure 5-3 La console Firebug après quelques essais de script

Remarquez que `console.error` génère une véritable erreur JavaScript, que Firebug détecte et comptabilise dans la barre d'état. Sachez par ailleurs que vous pouvez vous servir de ces méthodes avec une syntaxe de style `sprintf`, les codes de formatage possibles étant %s (texte), %d (nombre entier), %f (nombre à virgule) et même %o (lien vers l'objet qui sera passé en argument).

On trouve aussi une foule de méthodes permettant d'examiner les propriétés d'un objet (`console.dir`), d'afficher la pile d'appels courante (`console.trace`), ou de créer des niveaux d'indentation pour les messages, selon une habitude un peu primitive de débogage (`console.group` et `console.groupEnd`). On peut aussi facilement calculer

le temps écoulé entre deux endroits du code (`console.time` et `console.timeEnd`), et d'autres joyeusetés détaillées sur <http://getfirebug.com/console.html>.

C'est donc un grand nombre de possibilités qui sont sous-utilisées dans la console de Firebug... La figure 5-4 essaie d'illustrer avec concision quelques-uns de ces appels.

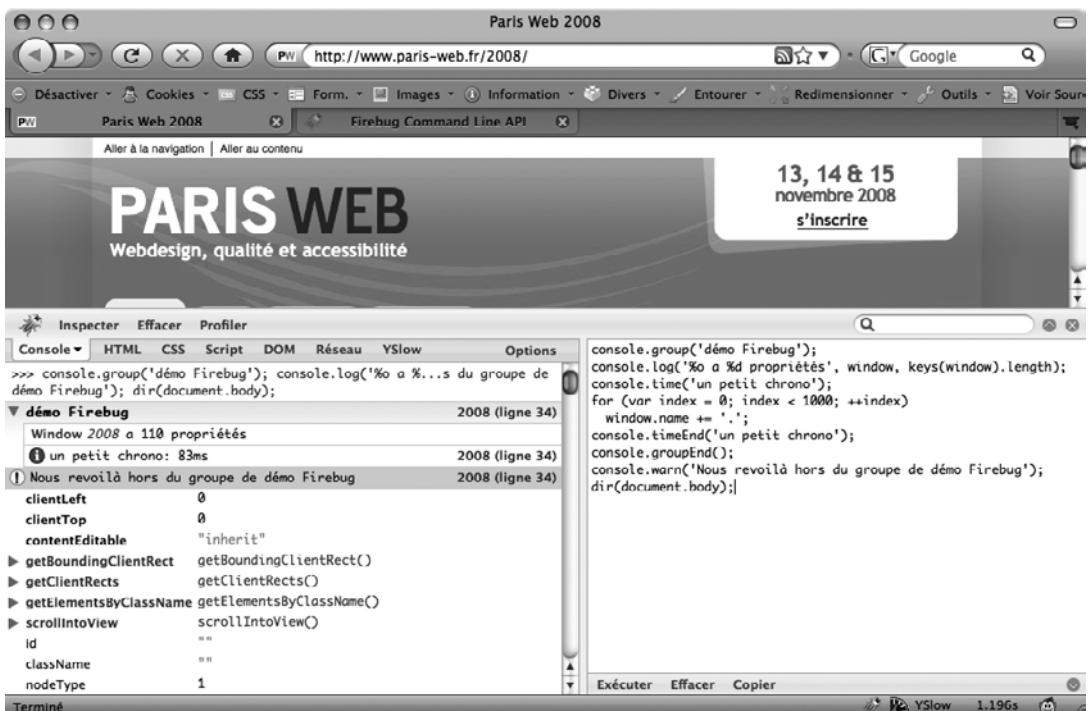


Figure 5-4 Quelques exemples parmi les nombreux appels possibles sur l'objet `console`

Remarquez l'utilité de fonctions comme `group` et `groupEnd`, qui créent un espace repliable groupant tous les affichages réalisés entre elles deux, ainsi que celle de `time` et `timeEnd`, qui nous évitent d'avoir à gérer nos propres obtentions, sauvegardes et soustractions d'horodatages...

Firebug propose par ailleurs un affichage un peu avancé des objets, tableaux et éléments du DOM, ainsi que certaines fonctions inspirées de Prototype, notamment `$` (qui récupère un élément par son ID) et `$$` (qui effectue une sélection CSS). Attention toutefois : si votre page a chargé des scripts avec ces fonctions, vous utiliserez les versions chargées par la page. Ainsi, sur le site de Paris Web qui utilise jQuery, la fonction `$` de Firebug est éclipsée. Voyons en revanche `$$` :

```
 $$('.liste')
```

On obtient un tableau d'éléments que Firebug représente intelligemment :

- Il en affiche, coloration syntaxique à l'appui, la balise, l'éventuel attribut `id=` et les classes CSS.
- Ces éléments sont ce que Firebug appelle des « hyperliens sur objet » (format %o des méthodes d'affichage de l'objet console), ce qui fait que :
 - D'une part, lorsqu'on les survole, ils se mettent en évidence sur la page elle-même, comme lorsqu'on utilise l'onglet *HTML* (voir l'annexe D) : des zones colorées indiquent le contenu propre, l'espacement interne (padding) et les marges.
 - D'autre part, cliquer sur ces hyperliens nous bascule justement dans l'onglet *HTML* et active immédiatement l'élément concerné, replacé donc dans son contexte.

Déboguer des scripts chargés

Tout script chargé, que ce soit un script dans un fichier externe (c'est bien !) ou mêlé au balisage (c'est moins bien !), est accessible depuis l'onglet *Script*. En haut de cet onglet, à côté de l'éternel bouton *Inspecter* qui nous bascule sur l'onglet *HTML* et permet de sélectionner rapidement des éléments dans la page par simple clic, se trouvent deux sections déroulantes.

La première, qui vaut par défaut *all*, permet de définir quelles sources de script nous intéressent ; dans la pratique, je vous conseille de passer sur *static*, parce que sinon, cela peut donner des listes invraisemblables ! Mais j'attire votre attention sur la seconde parce que, faute de flèche, on ne devinerait jamais que c'est une liste déroulante, et pourtant c'est elle, lorsqu'on clique dessus, qui nous donne accès aux différentes sources de script, et notamment aux fichiers externes et éléments `<script>` que vous allez pouvoir examiner individuellement, comme le montre la figure 5-5.

Le site de Paris Web utilise cependant une version compressée de ses scripts, ce qui ne va pas nous être d'un grand secours pour illustrer le débogueur JavaScript ; passons donc sur l'excellent *A List Apart* (<http://alistapart.com>), activons les onglets *Script* et *Réseau*, et choisissons dans la liste des scripts possibles `mint?js`.

Tout le script s'exécute au chargement de la page. Nous allons poser un point d'arrêt sur la ligne 4 en cliquant simplement à gauche du numéro de ligne dans la « gouttière » à gauche du code source (ce qui va faire apparaître un gros point rouge sombre), et recharger la page normalement. C'est là que notre point d'arrêt va se déclencher, comme le montre la figure 5-6.

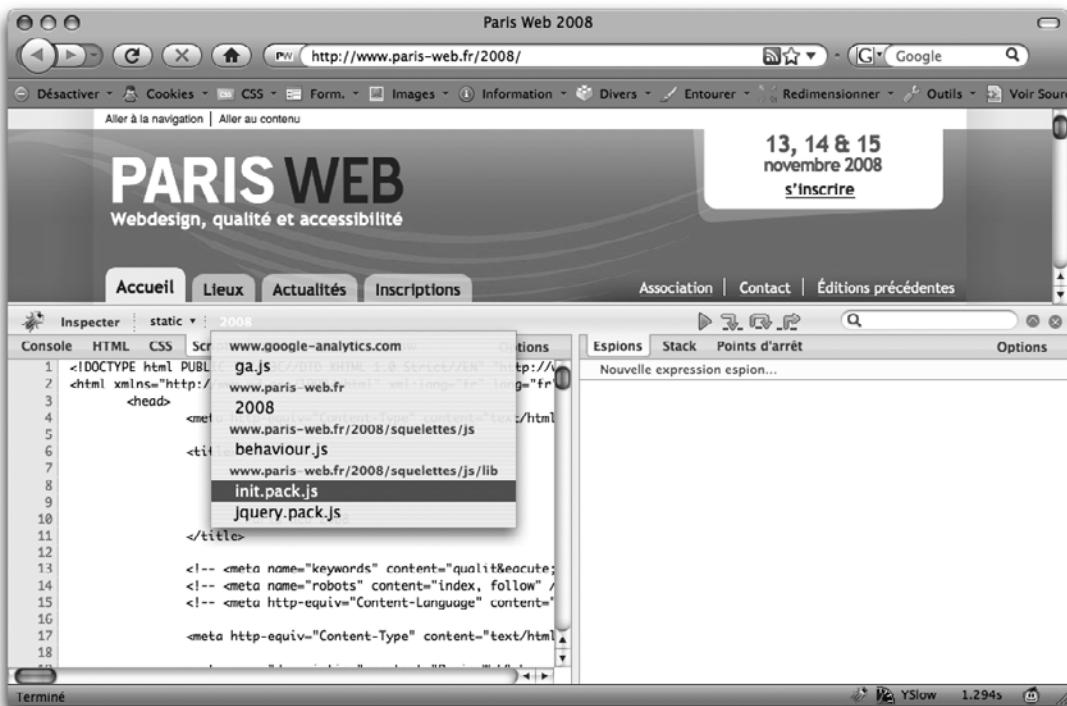


Figure 5–5 Choisir un script statique à examiner dans Firebug



Figure 5–6 Notre point d'arrêt se déclenche, nous sommes en mode débogage !

À gauche, le code source prêt à être exécuté comme bon nous semble (en général à l'aide de plusieurs modes de pas à pas). À droite, plusieurs sous-onglets :

- *Espions* (*Watches* en anglais) fournit les espions « automatiques » (variables locales existant au niveau de la ligne en cours d'exécution) et les espions permanents

(qu'on ajoute en cliquant simplement dans la ligne *Nouvelle expression espion...*). On y voit ici **this**, l'objet courant, doté manifestement d'une propriété `SI`, elle-même un objet ; et on y trouve, déjà prêtes quoique pas encore initialisées, les autres variables locales de notre fonction courante, `Mint.save()`. Firebug permet par ailleurs de survoler une variable dans un code en cours d'exécution pour obtenir sa valeur dans une infobulle.

- *Stack* représente la pile d'appels active, dont les derniers chaînons figurent aussi au-dessus de l'onglet *Script*, sous forme de liens vers les lignes en cours d'exécution dans les endroits correspondants du code. On est ici dans la fonction `save`, appelée depuis une fonction anonyme (l'initialiseur de la page).
- *Points d'arrêts*, enfin (*Breakpoints* en anglais), liste les endroits de nos scripts où nous avons déposé des points d'arrêt, avec leur état actif ou non (une case à cocher), le début de la ligne de code courante, le nom du script et le numéro de ligne, et une petite croix pour retirer définitivement le point d'arrêt. Firebug permet de définir une condition pour le déclenchement du point d'arrêt (cliquez du bouton droit sur la puce le représentant dans la gouttière), ainsi que de désactiver temporairement un point d'arrêt et de le réactiver plus tard (soit en allant dans cet onglet et en basculant la case à cocher, soit en utilisant *Shift+Clic* plutôt que *Clic* sur la puce rouge sombre correspondante à côté du code source).

Une fois en mode « débogage » suite à un point d'arrêt, nous avons à notre disposition les quatre actions usuelles de parcours du code, représentées par quatre icônes fléchées en haut du panneau Firebug, et associées à diverses touches suivant le système d'exploitation (sous Mac OS X avec un clavier français, les combinaisons sont parfois assez atroces, sous Windows ça va déjà mieux). Les actions sont :

- *Continuer*, qui relance le script jusqu'au prochain point d'arrêt éventuel sans s'arrêter entre-temps.
- *Pas à pas détaillé* (*Step Into* en anglais), qui exécute le code ligne par ligne, en entrant le cas échéant dans les fonctions appelées.
- *Pas à pas principal* (*Step Over* en anglais), qui exécute le code ligne à ligne mais sans entrer dans les fonctions appelées.
- *Pas à pas sortant* (*Step Out* en anglais), qui exécute le code jusqu'à sortir de la fonction courante, et repasse alors en pause dans le code appelant.

Ces fonctions sont bien connues de tout développeur ayant déjà utilisé un débogueur. Voyons-en une brève utilisation en pressant tranquillement, à quatre reprises, l'icône *Pas à pas principal* (ou la touche *F10* quand le système nous le permet). Nous allons ainsi exécuter plusieurs lignes jusqu'à avoir défini les variables `now`, `debug` et `path`, comme l'illustre la figure 5-7.



Figure 5–7 Quelques lignes plus loin après du pas à pas

Faites défiler le code et posez un point d'arrêt sur la ligne 50 (défilez ou tapez simplement #50 dans le champ de recherche, en haut à droite du panneau), qui démarre normalement par « var resource = ... » (évidemment, le script de Mint peut changer d'ici la sortie de cet ouvrage, mais je prends le risque). Cette ligne fait partie d'une fonction de rappel appelée suite à une requête Ajax. Pressez F5 pour laisser le script suivre son cours jusqu'à ce point d'arrêt (vu la vitesse des serveurs de Mint, ça devrait être instantané).

Regarder sous le capot d'Ajax

Nous voilà donc sur cette ligne 50, et je vous invite à ouvrir l'onglet *Réseau*. Vous y trouverez un appel Ajax en méthode GET vers le domaine courant, `alistapart.com`, avec son code de retour et sa durée. En cliquant sur la petite flèche à gauche, vous pourrez consulter les en-têtes de réponse mais aussi, plus bas, de requête, ainsi que le corps de la réponse dans l'onglet approprié.

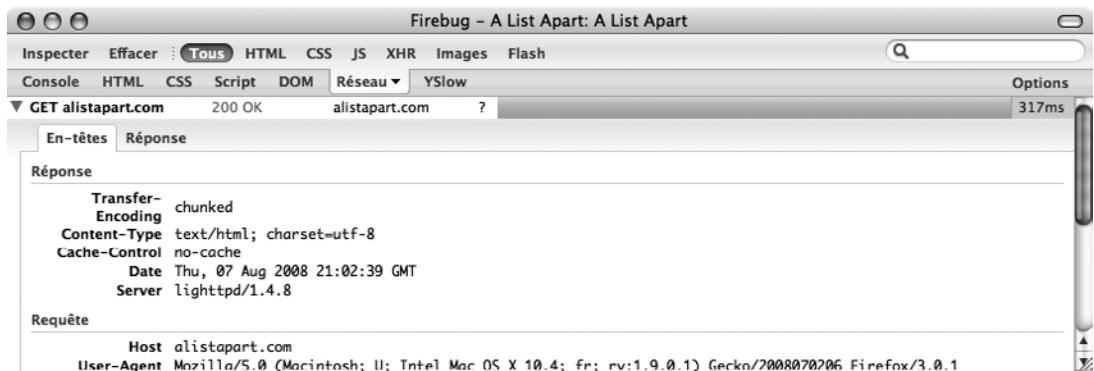


Figure 5–8 Le pistage des requêtes Ajax avec l'onglet Réseau de Firebug

Ce pistage des échanges Ajax est extrêmement précieux, et nous reviendrons dessus en détail au chapitre 6.

Examiner les performances avec le profileur

Dernier point qui peut être utile : la possibilité de *profiler* du code, c'est-à-dire de tracer le détail de l'exécution JavaScript pendant un intervalle de temps, pour obtenir un rapport par la suite. Lorsque vos scripts commencent à être sérieusement lents, c'est un bon endroit pour démarrer vos investigations, plutôt que de vous jeter à l'aveuglette dans des optimisations instinctives qui pourraient bien être du temps perdu.

Prenons le mini-site de Thomas Fuchs, le créateur de script.aculo.us, et voyons ce que ça donne ; allez sur <http://script.aculo.us/thomas>. Ouvrez Firebug et activez l'onglet *Script*, puis cliquez sur le bouton *Profiler* en haut du panneau. Utilisez maintenant la page : survolez les photos, cliquez sur quelques-unes pour déclencher les animations correspondantes. Quand vous avez fini, cliquez à nouveau sur *Profiler*. Vous obtenez alors un affichage extrêmement détaillé, similaire ce que montre la figure 5-9.



Figure 5-9 Le profileur de code de Firebug en action !

En cliquant sur les en-têtes de colonnes, vous pouvez trier les résultats pour savoir rapidement, par exemple, quelle fonction a pris au total le plus de temps brut (c'est-à-dire incluant les temps de ses sous-fonctions ; ici, si on fait exception de rapports erronés sur des enrobeurs créés par Prototype, ce serait `beat()`, une fonction clef du moteur d'effets du futur `script.aculo.us 2.0`), ou dure le plus longtemps en moyenne (ici ce bon vieux `show()`, à 12,5 millisecondes).

On peut aussi savoir quelles fonctions ont pris le plus gros du temps d'exécution (tri par défaut, ici `$A` et `setStyle`), ou ont été le plus appelées : après retrait des valeurs non pertinentes, on obtient ici `$A` (sur mon test, plus de 18 000 appels ! Heureusement que sa durée moyenne est de 0,01 ms...) et, sans surprise, `each` (sur mon test, près de 7 000 appels totalisant moins de 0,5 % du temps d'exécution).

Effectivement, quand votre page « rame », commencez par la profiler, cela vous donne de précieuses indications sur les pistes à suivre.

Firebug est décidément notre meilleur atout pour déboguer du JavaScript, mais lorsque tout marche bien sur Firefox et que ça casse sur MSIE, que faire ? Eh bien, le paysage est déjà moins rose...

Peaufiner sur IE6 et IE7

IE6 et IE7 totalisent encore en moyenne, à l'heure où j'écris ces lignes, 65 % de parts de marché au niveau mondial, et environ 55 % au niveau européen. À moins d'avoir la chance de développer pour un parc contrôlé avec uniquement des navigateurs conformes aux standards (idéalement Firefox 3, peut-être ?), il faut bien rendre notre page compatible avec IE7, et souvent aussi IE6, qui fait beaucoup plus de résistance que Microsoft ne le souhaiterait...

L'utopie d'un Windows pour deux

Laissez-moi d'abord piétiner un mythe : *non*, il n'est pas fiable d'avoir IE6 *et* IE7 cohabitant sur la même installation de Windows. En dépit des promesses d'outils comme MyIE, vous n'obtiendrez pas, dans la pratique, un comportement identique entre un IE6 qui « partage » son Windows avec un IE7, et un IE6 tout seul. Et inversement. Demandez donc à toutes les grandes agences web : ils ont essayé, ils en sont revenus fâchés.

Se pose aussi la question du système : IE7 sur XP n'étant pas rigoureusement identique, côté bugs, à IE7 sur Vista, lequel prendre ? Si j'étais vous, je prendrais IE7 sur XP Service Pack 2 ou 3, c'est le plus répandu et le plus rapide (Vista étant vraisemblablement

la pire version de Windows depuis WindowsMe, son adoption est très largement en deçà des espoirs de Microsoft, au point qu'ils ont dû réintroduire les licences OEM d'XP auprès des constructeurs, du jamais vu !)

Deux solutions, dès lors. On peut choisir de dédier deux machines physiques, l'une avec IE6 et l'autre avec IE7. Dans la pratique, ce n'est pas la panacée, parce que dès qu'on a plusieurs développeurs, ils sont en concurrence pour l'utilisation de ces machines, en particulier s'ils utilisent, comme souvent, une connexion de type « bureau à distance » (ce cher protocole RDP), laquelle déconnecte intempestivement l'utilisateur distant précédent...

Autre solution, que je vous recommande : investissez en mémoire vive (4 Go devraient suffire), récupérez un logiciel de virtualisation (par exemple VMWare, qui a des variantes gratuites, ou encore Parallels Desktop, un *must* absolu sur Mac OS X) et installez-y des machines virtuelles distinctes, une par version d'IE que vous souhaitez prendre en charge mais n'avez pas sur votre système principal. Ainsi par exemple, moi qui suis sur Mac OS X avec Firefox 3, j'en ai 3 qui sont toutes sur XP SP2 : une pour IE6, une autre pour IE7 et Firefox 2, et la dernière pour IE8.



Figure 5–10 IE6 et IE7 dans deux machines virtuelles... sur Mac OS X !

Suivant les détails de votre contrat de licence Windows, vous n'aurez pas forcément, je crois, besoin de 3 licences séparées : parfois le clonage de la première machine virtuelle, un XP SP2 IE6 simple, suffit pour produire la deuxième, qu'on passe sur IE7, qu'on clone à nouveau pour créer celle où on mettra IE8... Vérifiez tout de même les termes de votre contrat.

Le poids plume : Microsoft Script Debugger

Pour IE6 et IE7, le moyen le plus léger pour déboguer un peu son JavaScript est aussi gratuit : c'est le Microsoft Script Debugger. Il s'agit d'un outil un peu préhistorique quand même (la première version a déjà plus de 10 ans), téléchargeable *via* l'une de ces URLs dont Microsoft a le secret : <http://www.microsoft.com/downloads/details.aspx?familyid=2f465be0-94fd-4569-b3c4-dffdf19cccd99>. Vous pouvez aussi chercher « Microsoft Script Debugger » sur Google, ça marche bien...

C'est minuscule (654 Ko !) et uniquement en anglais. Il vous faudra par ailleurs une licence Windows authentique pour le télécharger, vu qu'il nécessite le Windows Genuine Advantage (lequel a de nombreux « faux positifs », bonne chance...).

Préparer le terrain avec une page à échec garanti

Commençons par nous faire une page qui va générer à coup sûr une erreur. Faites-vous un répertoire dédié, collez-y tout de même notre bon vieux `prototype.js`, puis saisissez le fichier `index.html` suivant :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      xml:lang="fr-FR">
  <head>
    <meta http-equiv="Content-Type" content="text/html;
          charset=iso-8859-15" />
    <title>Déboguer du JavaScript</title>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="demo.js"></script>
  </head>
  <body>

    <h1>Déboguer du JavaScript</h1>
    <form>
      <p><input type="button" id="btnBlam" value="Blam !"/></p>
    </form>

  </body>
</html>
```

Ensuite, posez le fichier JavaScript `demo.js` que voici :

```
function blam() {
    blamOnParagraphs(document.documentElement);
}

function blamOnParagraphs(baseNode) {
    if (baseNode.nodeName == 'P')
        baseNode.fakeMethod('blam');
    $(baseNode).childElements().each(blamOnParagraphs);
}

function initPage() {
    $('#btnBlam').observe('click', blam);
}

document.observe('dom:loaded', initPage);
```

Ce script n'est volontairement pas trivial afin d'illustrer des appels récursifs et de vous permettre de vous habituer à certaines piles d'appel « à la Prototype », comme celles exploitant `each` et les gestionnaires d'événements. Je vous encourage notamment à expérimenter ce script non seulement avec MS Script Debugger, mais aussi, par comparaison, avec Firebug et les autres outils que nous voyons dans ce chapitre.

La ligne qui va poser problème est celle où j'ai mis en exergue l'appel à `fakeMethod`, sur un élément DOM de type paragraphe (le seul de notre page). Les éléments du DOM n'ont évidemment pas de méthode `fakeMethod`, aussi cet appel va-t-il engendrer une erreur.

Configurer MSIE pour autoriser le débogage

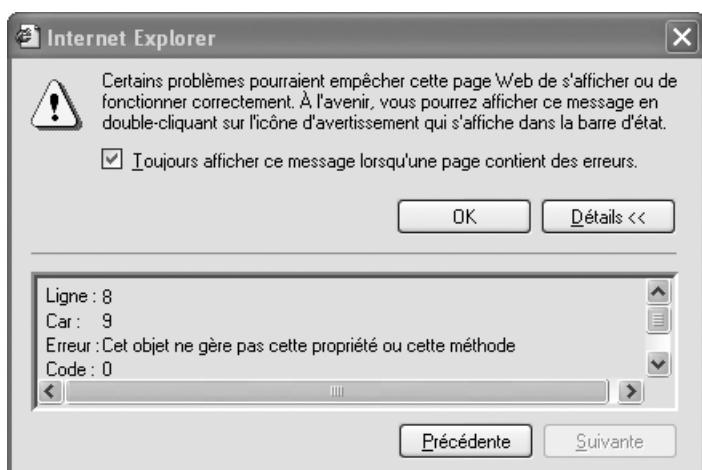
Par défaut, MSIE n'autorise pas des applications tierces à déboguer le JavaScript des pages visitées. Il ne prend même pas la peine d'afficher une notification pour chaque erreur de script, ce qui fait que nos pages ont l'air d'échouer « silencieusement », sans la moindre information utile.

Chargez notre page directement dans IE6 ou IE7, puis cliquez sur le bouton que nous avons mis en place.

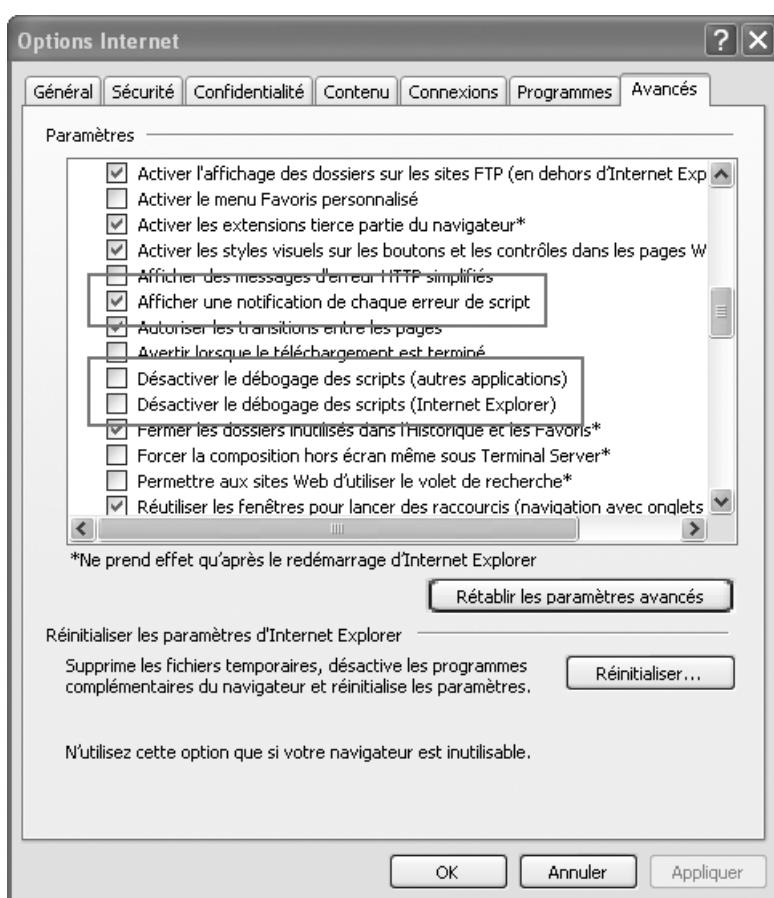
Lorsque l'option avancée *Afficher une notification de chaque erreur de script* est activée, vous obtenez juste une fenêtre bien peu informative, comme celle de la figure 5-11.

Figure 5–11

Les notifications d'erreur JS de MSIE : plutôt sommaires...

**Figure 5–12**

Les options à manipuler pour déboguer du JS sur MSIE



Rien ici ne nous permet de passer en mode de débogage ; afin d'assurer ces notifications et d'autoriser le débogage, nous allons devoir modifier certaines options, dont les libellés sont les mêmes sur IE6 et IE7. Ce sont :

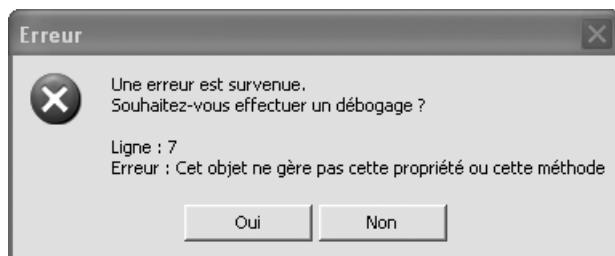
- *Afficher une notification de chaque erreur de script*, déjà évoquée, qu'il vous faut cocher.
- *Désactiver le débogage des scripts (autres applications)*, qu'il est préférable de décocher.
- *Désactiver le débogage des scripts (Internet Explorer)*, qu'il faut décocher.

La figure 5-12 montre ces options dans leur contexte.

Une fois MS Script Debugger installé, ces options configurées et MSIE redémarré, cliquer sur notre fameux bouton va plutôt afficher une boîte de dialogue comme celle de la figure 5-13.

Figure 5-13

MSIE nous propose de déboguer une erreur JavaScript



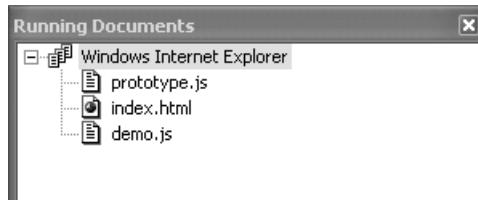
Si nous choisissons *Oui*, nous tombons sur un morceau de code interne à Prototype, en raison de la structure de `Enumerable#each`, qui contient un bloc `catch` dans lequel notre erreur est captive.

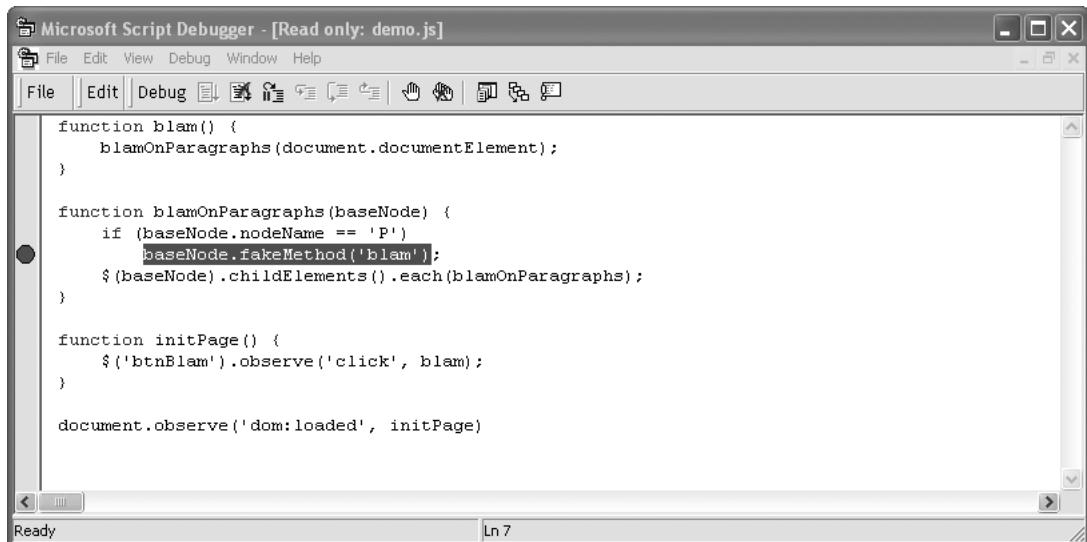
Les possibilités du débogueur

Pour illustrer un peu mieux les possibilités fournies, nous allons demander à continuer l'exécution (*Debug | Run* ou *F5*), puis lister les documents actifs (*View | Running documents*, ce qui donne l'affichage montré en figure 5-14) et choisir notre script `demo.js`. Nous allons y poser un point d'arrêt, ce qui n'est hélas pas faisable avec l'habituel clic dans la gouttière : il va falloir poser le curseur de texte sur la ligne en question (celle de l'appel à `fakeMethod`) et presser *F9* ou cliquer sur l'icône de barre d'outils correspondante (la main blanche). La figure 5-15 montre l'aspect final de notre code après ces manipulations.

Figure 5-14

La fenêtre des documents actifs dans MS Script Debugger





The screenshot shows the Microsoft Script Debugger interface. The title bar reads "Microsoft Script Debugger - [Read only: demo.js]". The menu bar includes File, Edit, View, Debug, Window, and Help. The toolbar contains icons for Open, Save, Cut, Copy, Paste, Find, Replace, and others. The main code editor window displays the following JavaScript code:

```
function blam() {
    blamOnParagraphs(document.documentElement);
}

function blamOnParagraphs(baseNode) {
    if (baseNode.nodeName == 'P')
        baseNode.fakeMethod('blam');
    $(baseNode).childElements().each(blamOnParagraphs);
}

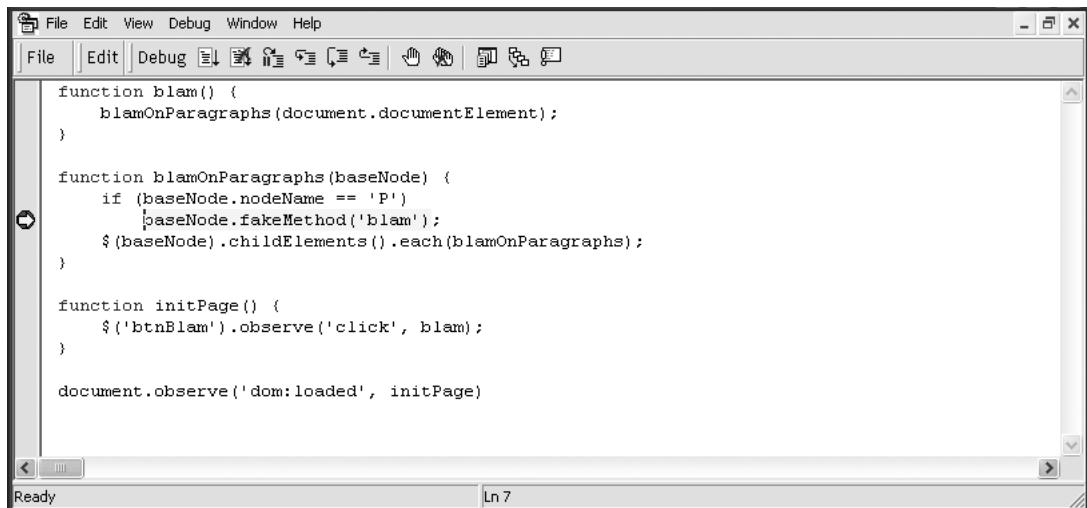
function initPage() {
    $('#btnBlam').observe('click', blam);
}

document.observe('dom:loaded', initPage)
```

A single red circular breakpoint marker is positioned at the start of the first function definition. The status bar at the bottom left says "Ready" and at the bottom right says "Ln 7".

Figure 5-15 Notre script demo.js avec son point d'arrêt

Cliquez à nouveau sur le bouton ; cette fois-ci, notre point d'arrêt se déclenche, comme sur la figure 5-16. Remarquez que vous ne disposez d'aucune fenêtre ou vue « espions » ; si vous voulez examiner le contexte, il va falloir l'explorer manuellement avec la fenêtre de commande (*View | Command window*), en tapant les expressions qui seront évaluées dès que vous appuierez sur *Entrée*, comme sur la figure 5-17.

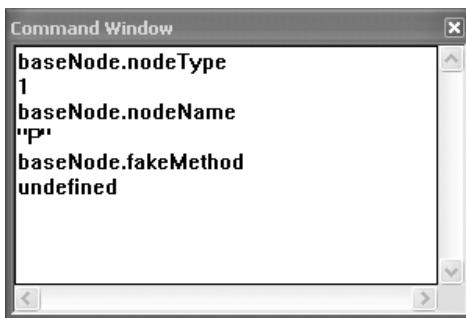


The screenshot shows the Microsoft Script Debugger interface, similar to Figure 5-15. The title bar reads "Microsoft Script Debugger - [Read only: demo.js]". The menu bar includes File, Edit, View, Debug, Window, and Help. The toolbar contains icons for Open, Save, Cut, Copy, Paste, Find, Replace, and others. The main code editor window displays the same JavaScript code as Figure 5-15. However, the first line of code now has a red circular breakpoint marker with a small black dot inside it, indicating that the debugger has stopped at this point. The status bar at the bottom left says "Ready" and at the bottom right says "Ln 7".

Figure 5-16 Notre point d'arrêt déclenché : le débogage peut démarrer...

Figure 5-17

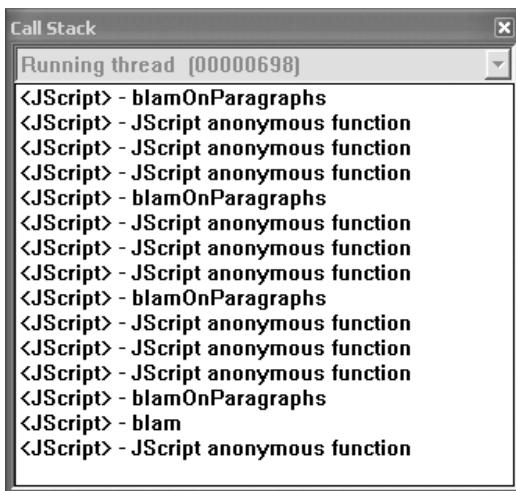
La fenêtre de commande, une console au rabais (mais c'est mieux que rien)



La pile d'appels est quant à elle visible avec *View | Call stack*, ce qu'illustre la figure 5-18. Double-cliquer sur un élément de la pile d'appels vous amène sur la ligne de code correspondante.

Figure 5-18

La pile d'appels ; on navigue dans le code à coup de double clics



Bien sûr, on dispose des possibilités habituelles d'exécution du code, accessibles au travers du menu *Debug*, des boutons de barre d'outils et de divers raccourcis qu'on retrouve pour la plupart dans les différents EDI Microsoft (VBA, Visual Studio, etc.) : Continuer (*Run / F5*), Arrêter le débogage (*Stop Debugging / Maj+F5*), Pas à pas détaillé (*Step Into / F8*), Pas à pas principal (*Step Over / Maj+F8*), Pas à pas sortant (*Step Out / Ctrl+Maj+F8*) et Bascule de point d'arrêt (*Toggle breakpoint / F9*).

MS Script Debugger n'est certes pas la panacée, loin s'en faut. On est des années-lumière derrière Firebug ou les autres débogueurs que nous allons voir ; mais quand on est coincé avec IE6 et IE7, et qu'on ne dispose pas d'une plate-forme poids lourd interfacée sur le débogage de script, cet outil reste notre meilleure carte...

Le poids lourd : Visual Web Developer 2008 Express Edition

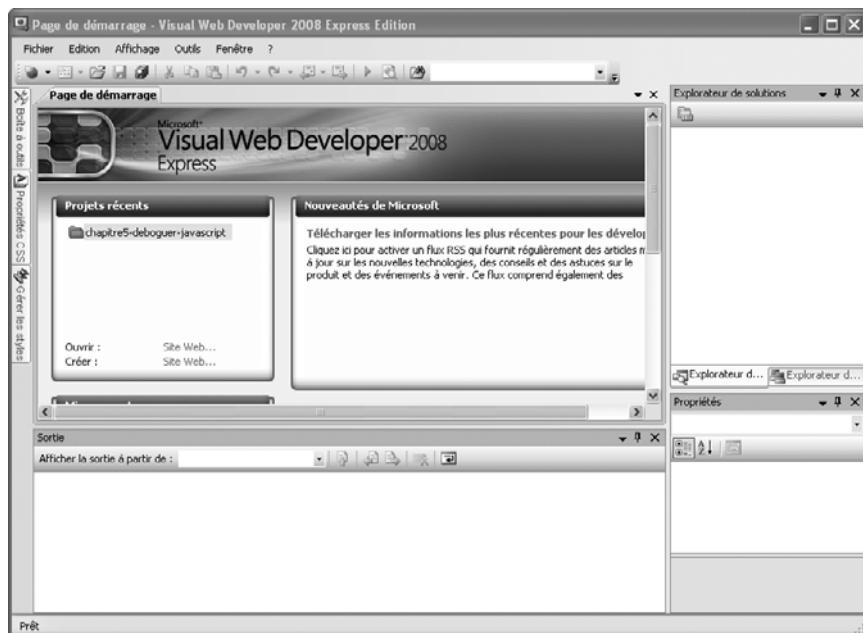
Si vous n'avez pas déjà un Visual Studio complet sur votre machine de développement, et ne pouvez vous satisfaire des fonctions rudimentaires de MS Script Debugger, vous pouvez tenter Visual Web Developer (VWD) 2008 Express Edition. Derrière ce nom à rallonge se cache une version « allégée » (à 1,4 Go, ça frise l'ironie...) de Visual Studio 2008, recentrée sur le développement web. Toute la partie serveur ne nous intéresse pas, mais c'est pour ses possibilités client, et spécifiquement de débogage de script, que l'outil peut nous servir.

VWD peut être téléchargé gratuitement (et sans montrer nécessairement patte blanche avec un Genuine Advantage) sur <http://www.microsoft.com/express/download/default.aspx>, le site officiel des séries Express. Il est disponible en français. On télécharge d'abord un installateur de 2,6 Mo environ, lequel téléchargera par la suite quelque 124 Mo (compter une dizaine de minutes sur une bonne connexion métropolitaine, et presque autant pour l'installation qui suit), pour une empreinte finale sur le disque de l'ordre de 1,4 Go ! Tout ça pour déboguer du JavaScript... Cependant, pour passer le cap de MS Script Debugger, ça vaut tout de même le coup...

L'installation est normalement sans embûches. Je vous recommande toutefois de décocher les trois installations complémentaires (MSDN Library, SQL Server, Silverlight), histoire de gagner du temps. Lorsque vous lancez VWD, vous tombez sur son écran d'accueil, comme le montre la figure 5-19.

Figure 5-19

L'écran d'accueil de Visual Web Developer 2008 Express Edition

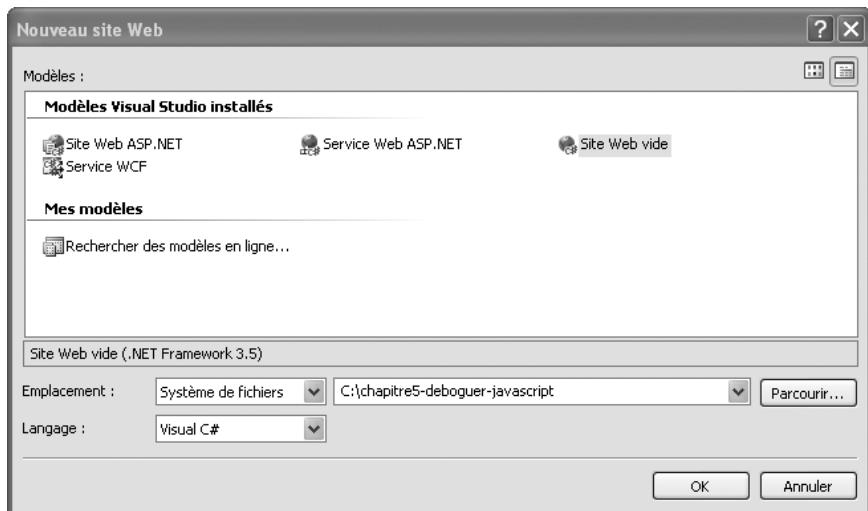


Voici maintenant le mode opératoire pour déboguer votre JavaScript alors que vous n'avez pas forcément travaillé avec VWD dès le début. C'est tout simple... une fois qu'on a trouvé comment.

- 1 Allez dans *Fichier | Nouveau site Web...* (*Alt+Maj+N*). Vous arrivez sur le dialogue de la figure 5-20.

Figure 5-20

Créer un nouveau site web calé sur notre dossier existant



- 2 Choisissez l'option *Site Web vide*, mais avant de valider, utilisez le bouton *Parcourir...* pour sélectionner le répertoire où vous avez placé vos sources. Validez.
- 3 VWD détecte que le dossier existe et n'est pas vide, et vous demande une confirmation (figure 5-21) ; choisissez l'option *Créer un site Web à l'emplacement existant* et validez.

Figure 5-21

Confirmer la création dans le dossier existant



- 4 Voilà, vous y êtes, comme en témoigne la figure 5-22.

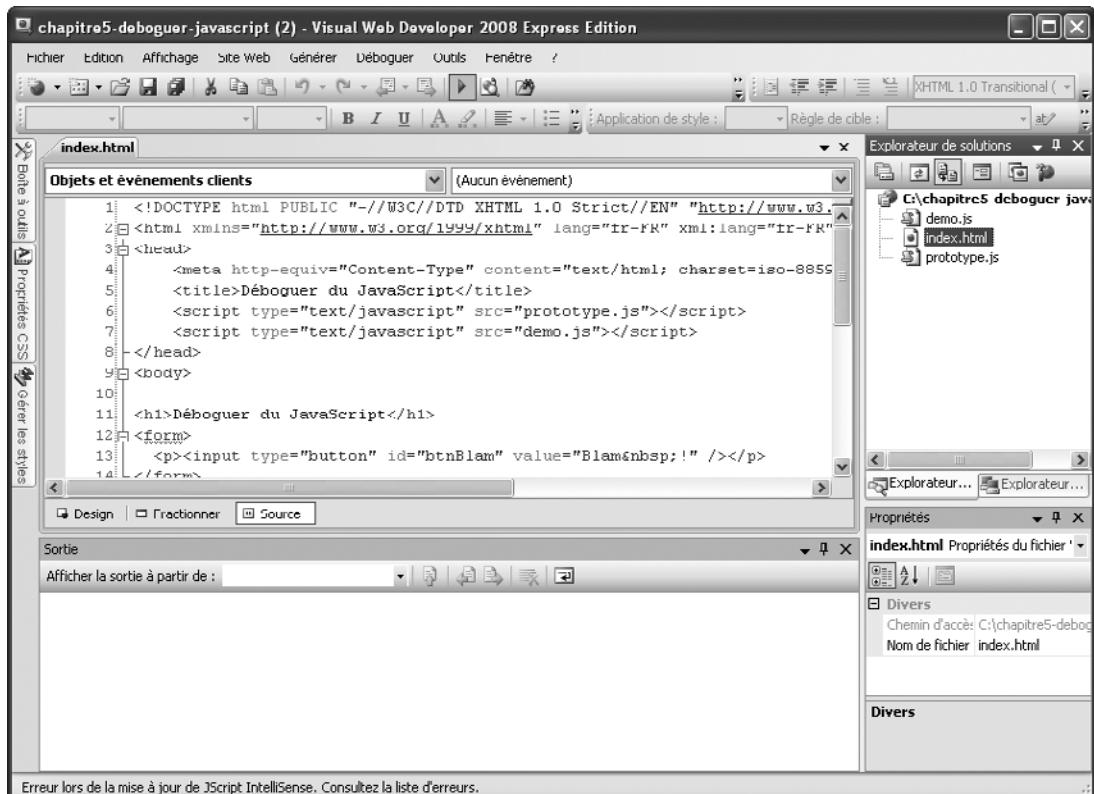


Figure 5-22 Nos fichiers « importés » dans le projet VWD, prêts à être débogués !

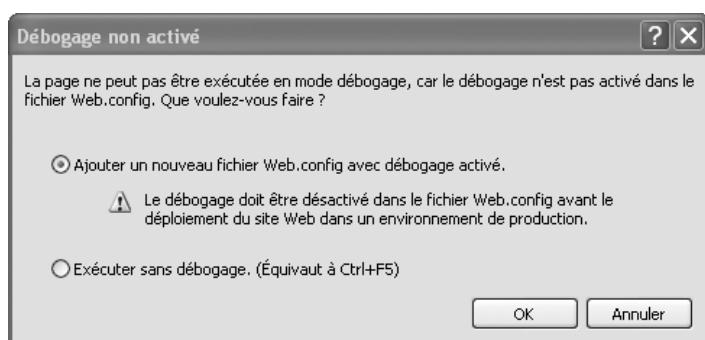
- 5 Cliquez alors sur la flèche verte dans la barre d'outils, ou choisissez *Déboguer | Démarrer le débogage (F5)*. Comme c'est la première fois, VWD va avoir besoin de créer une configuration d'exécution pour votre projet (un fichier `Web.config`), et vous demande confirmation qu'il faut bien y activer le débogage, comme sur la figure 5-23. Assurez-vous que la bonne option est cochée et validez.

Et voilà pour la partie générique ! Nous allons maintenant passer à notre page de test.

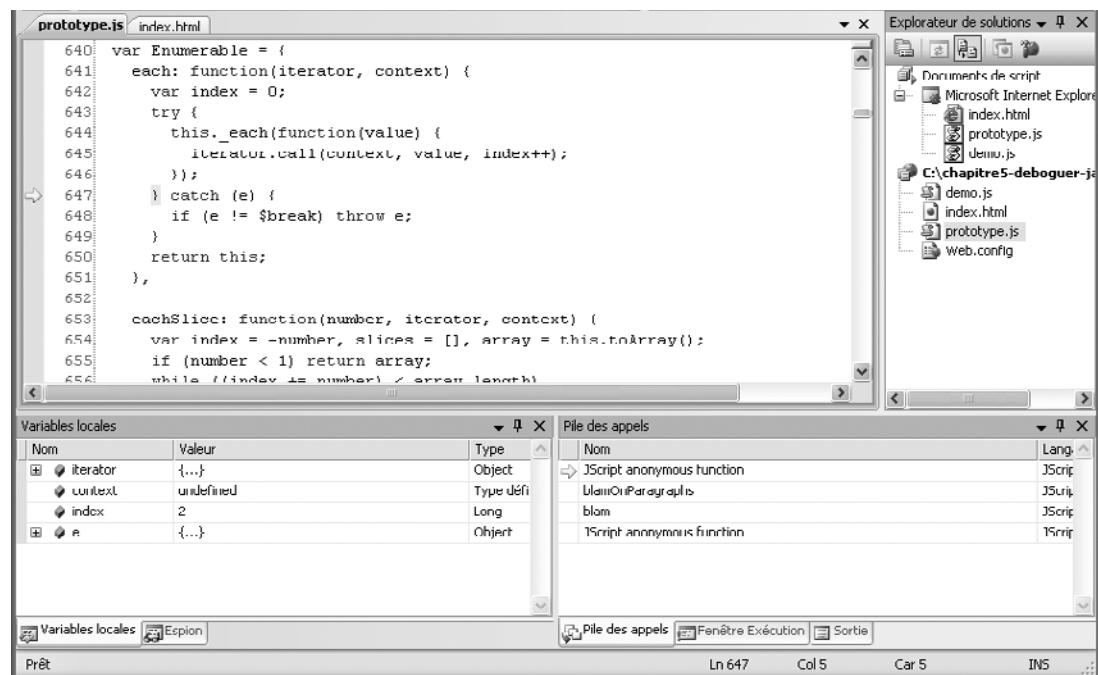
- 6 VWD affiche votre page dans un Internet Explorer lancé en mode débogage. Cliquez sur notre fameux bouton : VWD prend la main en affichant le dialogue de la figure 5-24. Choisissez le bouton *Arrêter* pour passer en mode pas à pas.

Figure 5–23

La première fois, confirmation de l'exécution en mode débogage

**Figure 5–24**

Passer en mode débogage depuis IE vers Visual Web Developer

**Figure 5–25** L'arrivée dans VWD suite à l'erreur que nous avons causée

- 7 Toujours pour les mêmes raisons, nous arrivons sur le bloc `catch` du `each` fourni par Prototype (figure 5-25). Dans le panneau *Pile des appels*, vous pouvez double-cliquer sur la fonction appelante `blamOnParagraphs`, ce qui nous emmène sur la ligne de code correspondante (figure 5-26).

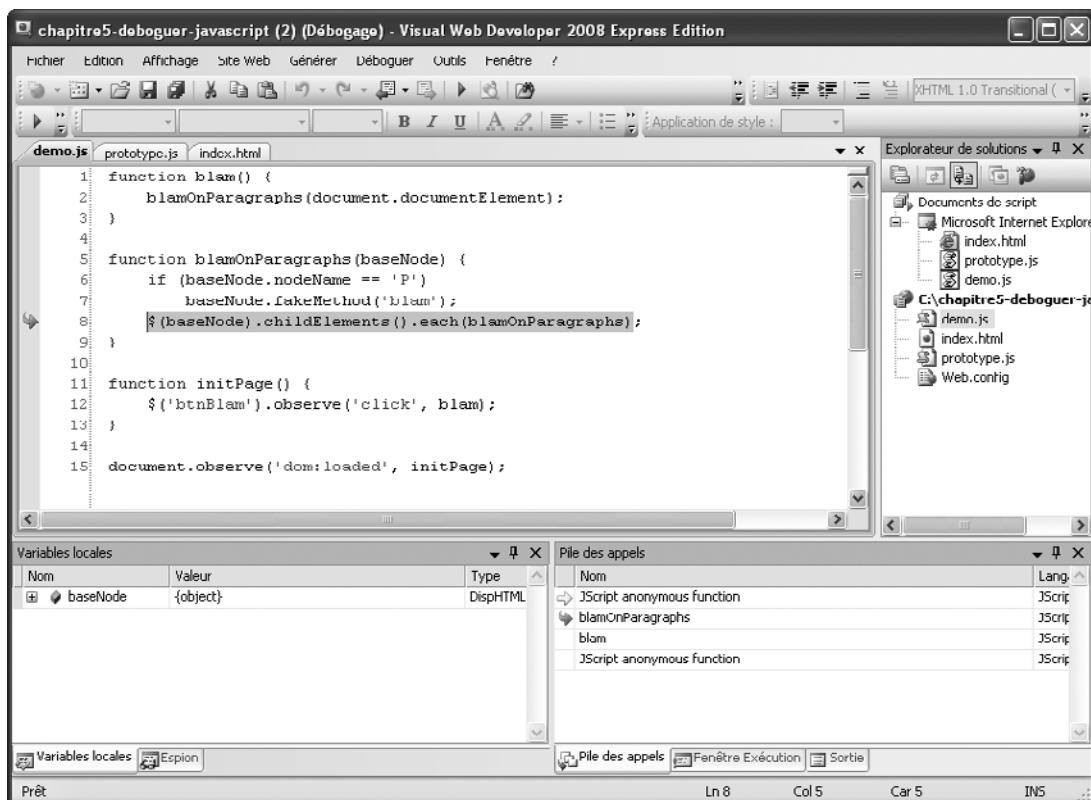


Figure 5-26 Navigation dans la pile d'appels

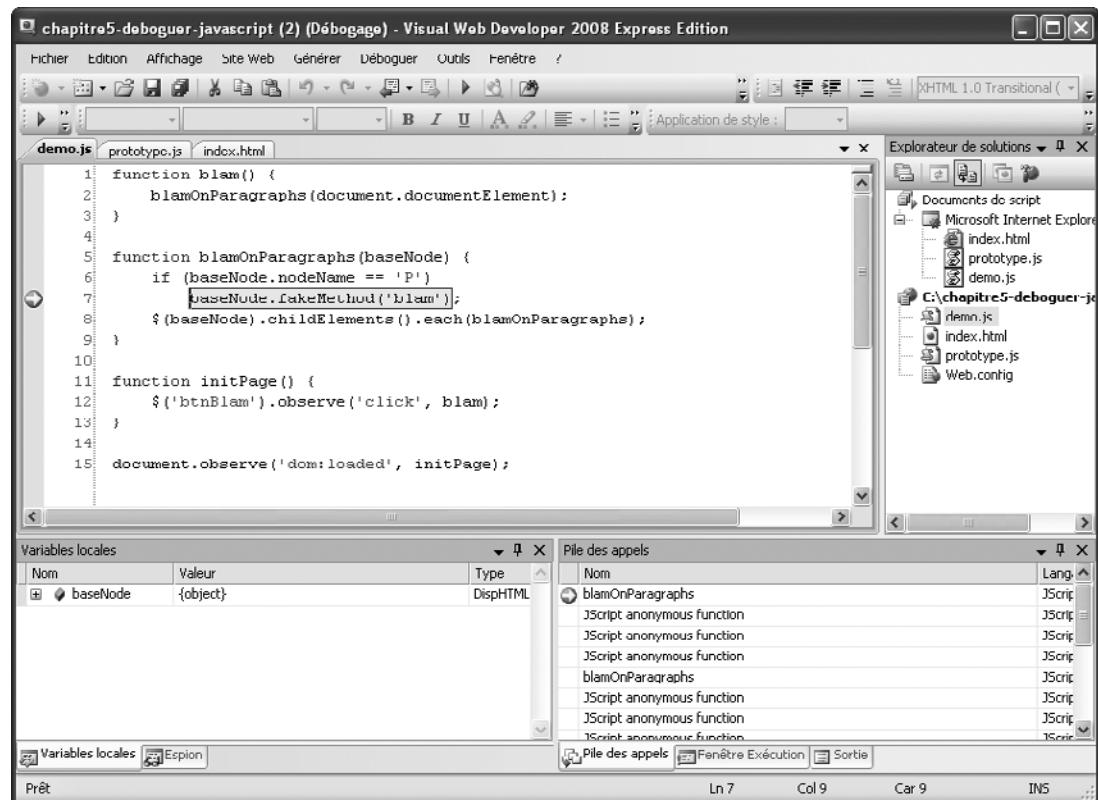
- 8 Sortons du pas à pas, avec *F5* par exemple. Visual Web Developer vous demande alors confirmation que vous voulez laissez repartir l'exception jusqu'à la page web elle-même (figure 5-27).
- 9 Revenez ensuite dans VWD et ouvrez le fichier `demo.js` dans le panneau *Explorateur de solutions* à droite ; vous pouvez y coller un point d'arrêt sur la ligne 7, comme toujours chez Microsoft en plaçant le curseur sur cette ligne et en pressant *F9*.

Figure 5–27

Continuer le débogage en repassant l'exception à notre page sur IE



- 10** Cliquez à nouveau sur le bouton dans la page, et vous arrivez dans l'éditeur qui présente l'aspect illustré par la figure 5–28. Vous remarquez la pile des appels, les variables locales, etc.

**Figure 5–28** Arrivée sur le point d'arrêt depuis IE

Comme vous pouvez le voir, c'est tout de même beaucoup mieux que MS Script Debugger, et finalement, du strict point de vue du débogage JavaScript, tout à fait satisfaisant.

Peaufiner sur IE8

Internet Explorer 8 est d'ores et déjà disponible en version de « beta-test » sur le site de Microsoft, tant pour Windows XP que pour Vista, à l'adresse <http://www.microsoft.com/windows/products/winfamily/ie/ie8/getitnow.mspx>. Attention, la remarque sur les multiples versions d'IE sur une même machine tient toujours : si vous installez IE8, vous pouvez dire au revoir à votre IE6 ou IE7 ; la désinstallation ne garantit pas que vous retrouvez un comportement identique à celui qui prévalait avant d'avoir mis IE8 sur la machine... Prévoyez donc une machine virtuelle à part ou un équivalent.

Tout comme IE7 a constitué une belle avancée dans la prise en charge de CSS, IE8 souhaite faciliter la vie des développeurs web au travers d'améliorations significatives dans la prise en charge du DOM tel que défini par le W3C, ainsi que du moteur JavaScript. Dans le même esprit, il fournit désormais des outils dédiés au développeur en direct, intégrés au navigateur. Ces outils se résument finalement à une version légèrement améliorée de la Internet Explorer Developer Toolbar (décrise en détail dans l'annexe D) couplée à un débogueur JavaScript. C'est ce dernier qui nous intéresse ici.

Contrairement au comportement existant sur IE6 et IE7, IE8 ne s'interface pas avec un débogueur externe, et n'affiche donc pas de message « voulez-vous déboguer ? » en cas d'erreur. Il faut explicitement entrer en mode débogage sur un document pour pouvoir ensuite manipuler ses scripts, poser des points d'arrêt, etc.

Chargez notre page à problèmes dans IE8 puis ouvrez les outils de développement en cliquant sur la petite flèche bleue à droite de la barre d'outils (il vous faudra peut-être redimensionner cette dernière pour la voir en entier) ou en choisissant dans les menus *Tools | Developer tools*. Prenez alors l'onglet *Script*. Dans la liste déroulante des fichiers en haut à gauche de l'onglet, vous ne voyez pour le moment que le document HTML ; c'est parce que le mode débogage n'est pas enclenché : activez-le avec le bouton *Start Debugging*. Dans la liste déroulante mise à jour, vous pouvez choisir `demo.js` et poser un point d'arrêt en cliquant dans la gouttière à gauche de la ligne appelant `fakeMethod`. La figure 5-29 montre le résultat obtenu.

À présent, cliquez sur notre bouton fauteur de troubles : le point d'arrêt se déclenche. Vous remarquez tout de suite la vue *Locals* à droite, illustrée sur la figure 5-30. Elle affiche automatiquement les variables locales et permet de naviguer à l'intérieur, ce qui nous est d'une aide précieuse.

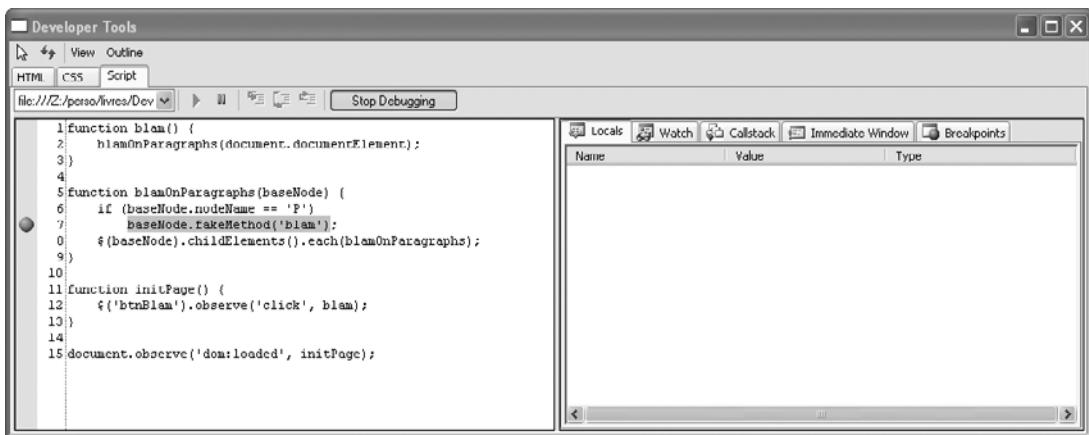


Figure 5–29 Notre point d’arrêt dans le débogueur JavaScript d’IE8

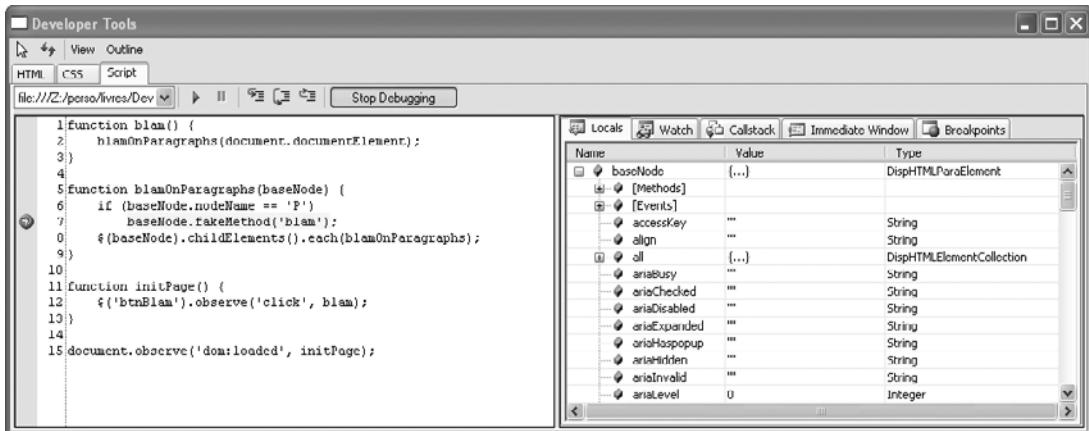


Figure 5–30 Les variables locales dans le débogueur JavaScript d’IE8

Des espions permanents peuvent être définis dans la vue *Watch*, la pile d’appels est affichée de façon un peu plus agréable dans *Callstack*, les points d’arrêt sont regroupés dans *Breakpoints* (mais ne sont pas conditionnels, bien qu’ils soient devenus désactivables), et enfin la vue *Immediate Window* permet la saisie monoligne ou multiligne de code JavaScript ! On a fait du progrès !

Petit conseil d’utilisation : ne désactivez *jamais* le mode débogage alors que vous êtes dans un pas à pas. Continuez l’exécution jusqu’à être revenu(e) au document avant de cliquer sur le bouton *Stop debugging*, sous peine de geler le navigateur la majorité du temps...

Peaufiner sur Safari

Safari 2 proposait un menu *Debug* « caché », qu'on activait en modifiant manuellement une préférence du navigateur, généralement depuis un terminal (`defaults write com.apple.Safari IncludeDebugMenu 1`). L'édition précédente de ce livre vous en parlait rapidement. Safari 3 a fait des progrès. Nous n'avons plus besoin de ce réglage, car Safari inclut désormais par défaut un menu *Développement* orienté développeur web, là où l'ancien menu *Debug* était plutôt orienté « développeurs de Safari »...

L'annexe D explore les parties HTML, CSS et DOM de ce menu ; nous nous intéresserons ici plus spécifiquement à ses capacités JavaScript, qui sont hélas un peu légères.

Une fois la page chargée, si vous cliquez sur le bouton, vous ne verrez pas l'erreur : il ne se passera simplement rien visuellement. En allant dans *Développement* | *Afficher l'inspecteur web* (*Cmd+Alt+I*), vous voyez les documents de la page et une catégorie *Scripts* qui liste les fichiers chargés. Notre `demo.js` affiche ici un nombre 1 sur fond rouge qui indique qu'une erreur est survenue dans ce fichier depuis le chargement de la page. En cliquant sur le nom du fichier, vous voyez le script avec, sous la ligne fautive, l'erreur en question. Voyez la figure 5-31.

The screenshot shows the Safari 3 Web Inspector interface. The left sidebar lists files under 'DOCUMENTS' (index.html) and 'SCRIPTS' (demo.js, prototype.js). The 'demo.js' file is selected. The main pane displays the source code of demo.js:

```
1 function blam() {
2     blamOnParagraphs(document.documentElement);
3 }
4
5 function blamOnParagraphs(baseNode) {
6     if (baseNode.nodeName == 'P')
7         baseNode.fakeMethod('blam');
8
9     $(baseNode).childElements().each(blamOnParagraphs);
10
11 function initPage() {
12     $('#btnBlam').observe('click', blam);
13 }
14
15 document.observe('dom:loaded', initPage);
```

A tooltip appears over the line `baseNode.fakeMethod('blam');` with the message: "Value undefined (result of expression baseNode.fakeMethod) is not object." Below the code pane, the 'Console' section shows a warning: "WARNING 1 error".

Figure 5-31 Une erreur JavaScript affichée dans le source par Safari 3

Ça s'arrête pratiquement là, parce qu'on n'a ni pas à pas, ni espions, ni points d'arrêt... En revanche, on a tout de même un outil *Console* (en bas à gauche de la fenêtre, accessible directement avec *Cmd+Alt+C*) qui est une fenêtre d'exécution directe JavaScript. Elle affiche aussi les erreurs avec une petite flèche naviguant vers le code fautif, et permet *via* sa ligne de saisie en bas de panneau d'exécuter du code directement, comme l'illustre la figure 5-32.

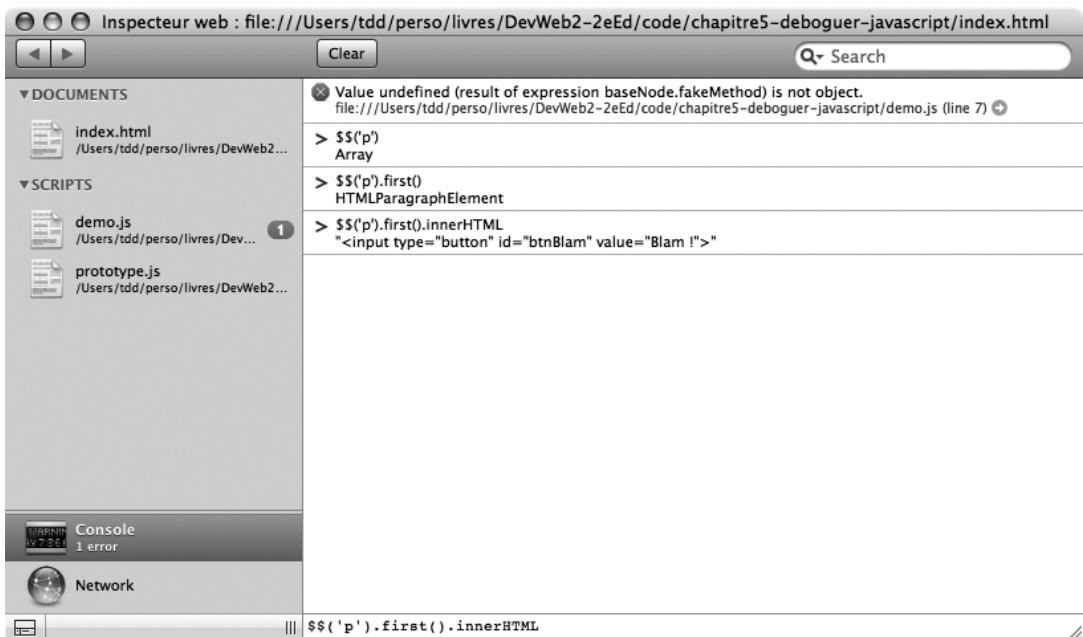


Figure 5-32 La console JavaScript de Safari 3

Peaufiner sur Opera

Depuis sa version 9.5, Opera dispose d'une série d'outils dédiés aux développeurs web, qui sont développés par une équipe constamment à l'écoute de la communauté. Ils répondent au nom collectif de Dragonfly, et Opera a fait le choix d'en faire un outil « en ligne » bien qu'intégré visuellement au navigateur. Du coup, si vous n'êtes pas connecté(e) à Internet, pas de Dragonfly !

Dans la pratique, Opera est un peu le dernier arrêt sur le chemin de la vérification de compatibilité : si votre script marche sur Firefox, IE et Safari, vous êtes plutôt tranquille, et devoir déboguer en urgence sans connexion ne vous arrivera probablement jamais...

On ouvre Dragonfly avec le menu *Outils* | *Avancé* | *Outils de développeur*, ou la combinaison *Ctrl+Alt+I*. La figure 5-33 montre l'aspect initial de l'outil.



Figure 5-33 Dragonfly chargé sur notre page, qui liste le fichier et les scripts

Ça tâtonne encore un peu, c'est moins élégant qu'ailleurs, mais ça fonctionne. En choisissant le script `demo.js` et en posant un point d'arrêt sur la ligne 7 par un simple clic dans la gouttière, on peut finalement cliquer sur notre cher bouton *Blam!* et arriver en pas à pas dans le script, comme on peut le voir sur la figure 5-34.

L'outil propose de nombreux onglets et des fonctionnalités avancées ; si l'onglet *Inspection* est assez familier, en listant les variables locales et automatiques (figure 5-35), et si la ligne de commande est assez classique, on trouve en furetant une pléthore de possibilités, comme le laisse prévoir la boîte de dialogue *Settings* accessible à travers l'icône de configuration en bas à droite de l'outil, et dont la figure 5-36 liste les catégories...



Figure 5–34 Un point d'arrêt sur notre script, dans Opera Dragonfly

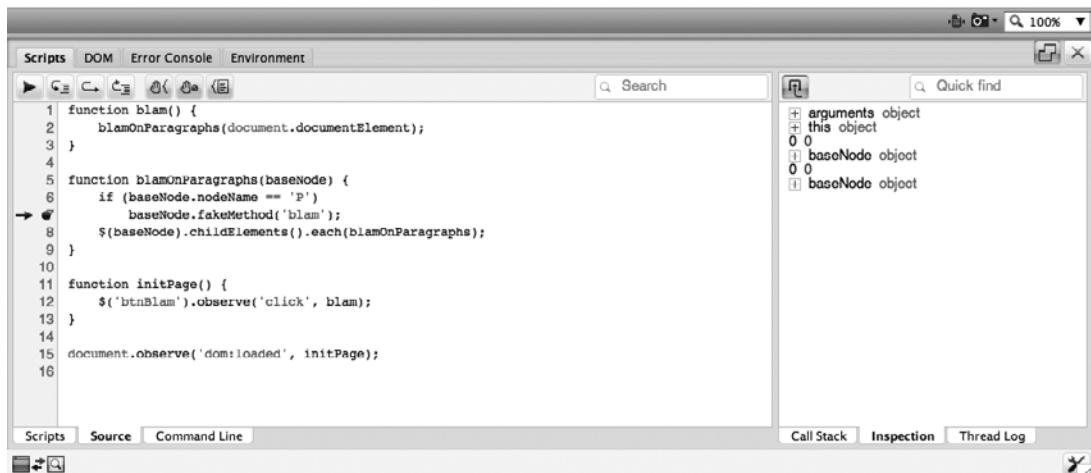


Figure 5–35 Le panneau Inspection de Dragonfly : variables locales et automatiques

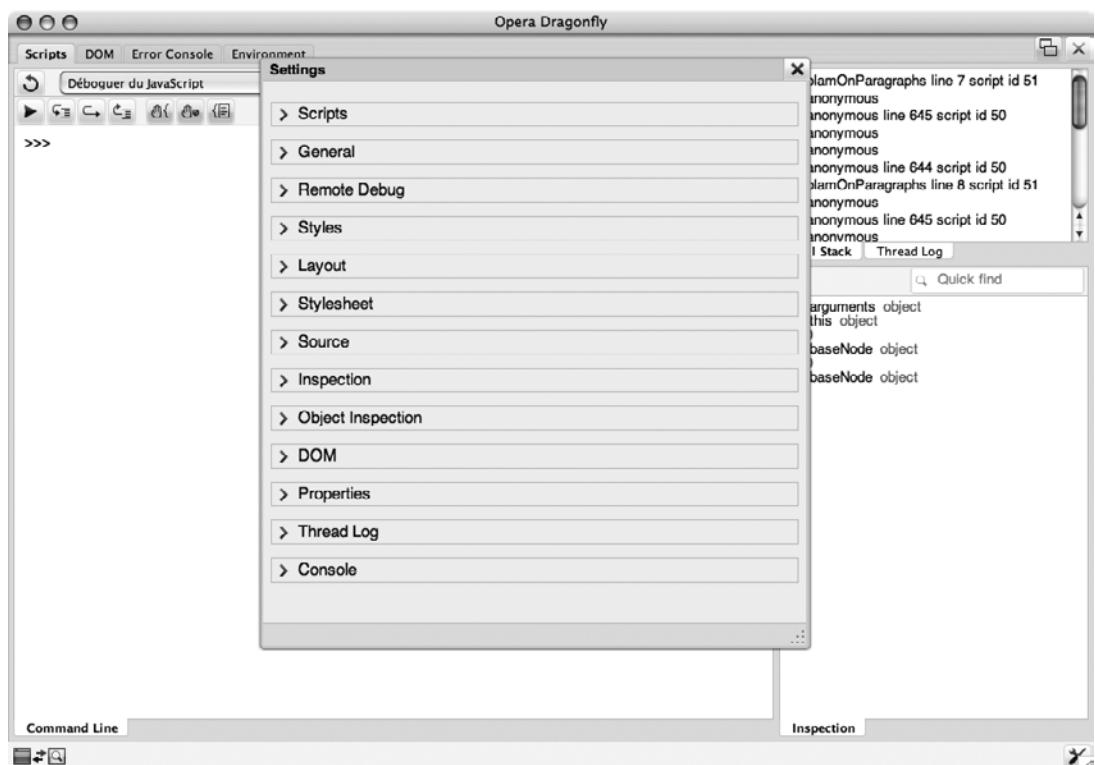


Figure 5–36 Les catégories de configuration de Dragonfly. Il y a de quoi faire !

Pour aller plus loin

Sites

- Firebug, le couteau suisse du développeur Web 2.0, est lui aussi une extension : <https://addons.mozilla.org/firefox/1843/>. Le site officiel fournit beaucoup d'exemples et de documentation : <http://getfirebug.com/>.
- On retrouve pas mal d'informations sur Safari à destination des développeurs web sur <http://developer.apple.com/internet/safari/>.
- Opera fournit un contenu extrêmement riche pour les développeurs web sur <http://dev.opera.com/>.

DEUXIÈME PARTIE

Ajax, ou l'art de chuchoter

Jusqu'ici, nous avons examiné les arcanes de JavaScript, et celles du DOM, qui se trouve au cœur de la manipulation dynamique des pages. Ensuite, nous avons présenté l'essentiel de la bibliothèque Prototype, qui rend nos codes JavaScript plus portables, mais surtout plus courts, plus simples, plus expressifs et tellement plus élégants.

Comprendre et maîtriser ces piliers du Web 2.0 constitue un préalable nécessaire à l'apprentissage de cet univers technologique dont Ajax est le sujet phare. Nous avons déjà étudié dans le détail le « J ». Restent les aspects « Asynchronous » et « XML ». Dans les chapitres qui suivent, nous allons découvrir puis explorer en profondeur les composantes techniques concernées.

On commencera par examiner la technologie « toute nue » au travers de sa clef de voûte, l'objet `XMLHttpRequest`, qui réalise des requêtes HTTP asynchrones pour le compte de scripts dans la page, et nous permet ensuite de traiter la réponse, au besoin en tant que document XML.

Bien comprendre les rouages de cet objet est indispensable pour éviter le risque d'une dépendance vis-à-vis de l'un ou l'autre des nombreux frameworks disponibles autour d'Ajax. Nous verrons que Prototype est toujours là pour nous faciliter grandement la tâche, tant pour des utilisations triviales que pour des manipulations plus avancées.

Nous découvrirons aussi quelques autres frameworks, au premier rang desquels l'incontournable `script.aculo.us` de Thomas Fuchs. Non content d'avoir créé une bibliothèque portable d'effets visuels spectaculaires (lesquels, employés à bon escient, augmentent considérablement la qualité de notre ergonomie), Thomas nous offre une panoplie de solutions toutes prêtes pour les emplois les plus courants d'Ajax, comme la complétion automatique.

6

Les mains dans le cambouis avec XMLHttpRequest

L'objet `XMLHttpRequest` est responsable, à lui seul, des aspects asynchrones et XML d'Ajax. C'est lui qui va effectuer des requêtes HTTP internes, invisibles, et asynchrones (s'exécutant en arrière-plan sans bloquer l'exécution des scripts ou geler la page) à une couche serveur. C'est lui aussi qui va traiter la réponse, éventuellement en tant que document XML.

Nous allons d'abord examiner ensemble l'anatomie d'une conversation Ajax entre une page web et la couche serveur. Nous installerons ensuite une couche serveur, justement, afin de pouvoir réaliser nos tests. Après un petit rappel historique sur `XMLHttpRequest`, nous rentrerons dans les détails techniques de son utilisation, et examinerons ensuite les différents types de réponse qu'on peut renvoyer, avec leurs intérêts respectifs.

Anatomie d'une conversation Ajax

Une conversation Ajax est constituée d'un ou plusieurs échanges, généralement asynchrones, entre la page web (au travers de son code JavaScript) et une couche serveur. On verra d'ailleurs au chapitre 9 que cette couche serveur peut être extrêmement variée, et n'a en rien l'obligation de résider sur le même serveur que vos pages.

Chaque échange suit la séquence que voici :

- 1 Création (ou réutilisation) d'un requêteur.
- 2 Association d'un gestionnaire d'état (une fonction qui nous appartient et qui va notamment traiter la réponse quand celle-ci arrivera).
- 3 Définition de la requête : mode de synchronisation, méthode HTTP (GET, POST, etc.), URL destinataire, paramètres et données éventuels.
- 4 Envoi de la requête.
- 5 Invocations du gestionnaire d'état au fil du cycle de vie de la requête, en particulier aux stades suivants : lancement, réceptions de parties de réponse, fin de réception de réponse.
- 6 Lorsque la réponse a été complètement reçue, traitement de celle-ci par notre code JavaScript (par exemple, insertion d'un nouvel élément dans une liste, affichage de suggestions pour un champ de saisie, affichage d'une portion d'image supplémentaire).

Il est important de bien comprendre qu'entre les étapes 4 et 5, la page « vit sa vie », ainsi que ses codes JavaScript. Elle n'est en aucun cas bloquée. Il est toujours possible d'utiliser une requête synchrone, mais cela bloque justement la page le temps du traitement, et réduit à néant l'intérêt du système ! En asynchrone, l'utilisateur conserve la possibilité d'interagir avec elle. On verra que cet aspect, qui constitue l'avantage fondamental de la technologie, n'est toutefois pas sans dangers ergonomiques et nécessite souvent une conception intelligente de l'interface.

Dans ce chapitre, nous aurons l'occasion d'examiner et de manipuler toutes ces étapes en détail, mais il va de soi que dans une utilisation industrielle, sur des projets réels et potentiellement lourds, on ne saurait s'encombrer des détails à chaque utilisation. Les chapitres suivants nous montreront comment tirer parti de frameworks répandus pour masquer l'apparente complexité du processus et nous concentrer sur la partie fonctionnelle.

Un petit serveur pour nos tests

Afin de pouvoir tester du code Ajax, nous devons avoir mis en place une couche serveur. En termes concrets, il nous faut donc, avant toute chose, un logiciel serveur HTTP installé, configuré et en cours d'exécution. Le tout doit résider soit sur notre propre machine, soit sur une machine joignable depuis la nôtre, puisque nous utiliserons notre navigateur pour visualiser nos pages de démonstration.

Mais ce n'est pas tout. Si nous nous arrêtons là, cela impliquerait que les contenus renvoyés par ce serveur HTTP soient figés, ce qui réduit grandement l'intérêt des tests et démonstrations que nous pourrions mener.

Il faut donc aussi que ce serveur prenne en charge un langage permettant de renvoyer un contenu dynamique. Le choix qui semble évident, pour cela, est PHP, en raison de sa popularité, de l'abondance de documentation et de sa prise en charge par virtuellement tous les hébergeurs. Et pourtant, ce n'est pas le choix que nous retiendrons. Voici pourquoi :

- 1 Nous ne souhaitons pas lier la possibilité de mener les tests à la disponibilité d'un compte et d'un espace de stockage chez un hébergeur. L'argument de la prise en charge est donc nul.
- 2 Nous ne souhaitons pas lier la facilité de mise en place d'un serveur de test à la plate-forme. Bien que PHP soit facile à utiliser à des fins de test sous Windows, notamment avec des produits comme EasyPHP, il nécessite une configuration élaborée sous Linux et Mac OS X.
- 3 Nous souhaitons éviter les éventuels conflits de ressource entre le serveur de test et des services déjà déployés. Ainsi, si vous êtes développeur (le simple fait que vous lisiez cet ouvrage signifie probablement que vous êtes développeur web, par exemple), vous avez peut-être déjà des serveurs MySQL, IIS ou Apache installés et opérationnels sur votre machine.
- 4 Nous souhaitons permettre aux lecteurs n'ayant pas une expérience préalable d'un langage côté serveur de s'y retrouver sans trop de mal dans nos exemples. Le langage retenu se doit donc d'être le plus lisible possible, ce qui n'est pas vraiment le cas de PHP, en particulier pour un néophyte.
- 5 La configuration nécessaire à nos tests doit être la plus simple possible. En ce qui nous concerne, il s'agit simplement d'associer à certaines URL du serveur des morceaux de code aptes à générer du contenu dynamique.
- 6 Le lancement et l'arrêt du serveur, ainsi que le suivi de son exécution, doivent être simplifiés au maximum. Ici, les serveurs classiques, même emballés dans une interface centralisée comme EasyPHP, sont déjà trop complexes, en particulier si la configuration pose problème. Le suivi de l'exécution, quant à lui, est tout simplement absent de tels outils, nécessitant des commandes tierces comme le `tail` d'Unix, ainsi que la connaissance de l'emplacement des fichiers de journalisation.
- 7 Plus subjectivement, j'estime qu'à technologie de pointe, langage de pointe ! Des possibilités comme PHP, JSP ou ASP reposent sur des langages rigides ou empreints d'idées anciennes.

Pour toutes ces raisons, j'ai décidé d'utiliser Ruby.

Pas de panique ! Peut-être ce langage ne vous dit-il rien, ou peut-être avez-vous déjà aperçu quelques bribes de code Ruby qui vous auraient semblé exotiques. Toutefois, j'estime qu'il s'agit là d'un choix judicieux pour ce chapitre, à de nombreux titres. En miroir aux considérations listées précédemment, les principales raisons de mon choix sont :

- 1 Ruby est très facile à installer, tant sur Windows que Linux ou Mac OS X. L'installation fournit une large bibliothèque standard d'objets et de services, dont un serveur léger HTTP tout équipé, apte bien sûr à exécuter du code Ruby pour générer du contenu.
- 2 WEBrick, le serveur HTTP fourni dans la distribution standard de Ruby, est très facile d'emploi. Il nous suffira de quelques lignes de code très simples pour décrire sa configuration, y compris l'association entre des URL et nos fonctions.
- 3 Lancer le serveur se fait d'une courte saisie dans la ligne de commande. L'arrêter est l'affaire d'un simple `Ctrl+C`. Le suivi de l'exécution a lieu automatiquement dans la console, sans avoir à chercher de fichiers de log.
- 4 Puisque le serveur est sur notre propre machine, nul besoin d'un compte chez un hébergeur, ni de procédures fastidieuses de déploiement sur ce compte.
- 5 Ruby est un langage très facile à apprendre et à lire. Et même si l'objet de ce chapitre n'est pas de vous enseigner les rudiments du langage (il ne s'agit après tout ici que d'un outil au service de l'apprentissage des mécanismes Ajax), vous verrez que le code parle de lui-même (et quand ce n'est pas le cas, les explications nécessaires sont courtes).
- 6 Ruby est un langage moderne, 100 % objet et très flexible, extrêmement productif, conçu d'après les leçons tirées de la mise en œuvre des langages récents.
- 7 Les principaux acteurs d'Ajax, notamment les auteurs des bibliothèques Prototype et script.aculo.us, sont très impliqués dans le framework Ruby on Rails (RoR), écrit intégralement en Ruby. Les deux technologies, qui contribuent à « élargir le champ des possibles » (*push the envelope*), ne sont donc pas sans rapport.

Installation de Ruby

Commençons donc par installer le nécessaire sur votre machine ! Rassurez-vous, il ne s'agit pas ici d'installer un énorme système de développement ; rien de commun avec Visual Studio.NET ou simplement le JDK de Sun. La taille varie suivant la plate-forme, mais dans tous les cas, elle est bien inférieure à 100 Mo.

Sous Windows

Un excellent projet est tout spécialement dédié aux développeurs Ruby sous Windows : l'installateur Ruby « en 1 clic ». Il s'agit d'un projet Open Source disponible sur <http://rubyinstaller.rubyforge.org>.

L'installateur met en place les éléments suivants :

- Une version très récente de Ruby (à l'écriture de ces lignes, la 1.8.4).
- L'éditeur libre SciTE (<http://www.scintilla.org/SciTE.html>), qui fournit de quoi travailler pour les développeurs souhaitant un éditeur puissant sans pour autant utiliser un EDI (environnement de développement intégré).
- L'EDI FreeRIDE (<http://freeride.rubyforge.org/wiki/wiki.pl>), un environnement complet pour travailler avec Ruby, qui fournit les fonctionnalités habituelles de ce type de produit : coloration syntaxique, complétion de code, modèles, exécution, débogage, etc.
- De nombreuses bibliothèques et outils Ruby couramment utilisés. On y trouve entre autres Rake, l'outil de *make* dans l'univers Ruby ; et le système de bibliothèques RubyGems, qui formalise les bibliothèques Ruby.
- La version HTML Help de l'ouvrage de référence, *Programming Ruby*, écrit par Dave Thomas et Andy Hunt, les « Pragmatic Programmers », qui ont popularisé Ruby en Occident. Attention toutefois, il s'agit de la première édition, pour Ruby 1.6. Une seconde édition, massivement améliorée et complétée pour la version 1.8, est disponible (au format papier, PDF colorisé, ou les deux). Elle est préférable si vous souhaitez découvrir Ruby plus avant :
<http://pragmaticprogrammer.com/titles/ruby/index.html>.

Hormis Ruby lui-même, tous ces composants sont optionnels. L'installateur est téléchargeable ici : http://rubyforge.org/frs/?group_id=167. Une fois le téléchargement terminé, voici les étapes à suivre pour l'installation :

- 1 Lancez le programme d'installation (par exemple, `ruby184-20.exe`).
- 2 Choisissez Next, puis à l'écran suivant I Agree.
- 3 Cochez European Keyboards (active le symbole € dans l'interpréteur interactif `irb`), quant au reste, libre à vous d'installer ce que vous voulez (l'espace disque total requis variera entre 83 et 87 Mo à l'heure où j'écris ces lignes). Sélectionnez ensuite Next.
- 4 Pour le chemin d'installation, C:\ruby est très bien, mais si vous souhaitez le changer abstenez-vous de toute espace dans le chemin retenu, afin d'éviter les problèmes potentiels. Cliquez sur Next.
- 5 Changez le nom du groupe de programmes à quelque chose de plus générique, par exemple Ruby. Choisissez ensuite Install.

6 L'installation s'exécute et prend un peu de temps car elle comporte de très nombreux petits fichiers.

7 En fin d'installation, choisissez Next. Décochez Show Readme et cliquez sur Finish.

Et voilà !

Jetez un œil au groupe de programmes Ruby : il propose de nombreuses ressources de documentation, le serveur de documentation des *gemmes* (paquets Ruby) et fxri, outil regroupant un moteur de recherche dans l'aide en ligne (outil en ligne de commande : ri) et l'interpréteur interactif de Ruby (outil en ligne de commande : irb).

Pour information, les modifications apportées aux variables d'environnement sont les suivantes :

- INPUTRC pour autoriser le symbole € dans irb.
- PATH inclut désormais C:\ruby\bin (adapté au chemin que vous avez choisi, évidemment), pour avoir accès aux nombreux programmes fournis : ruby, ri, irb, rdoc, gem mais aussi iconv et fxri.
- PATHEXT pour autoriser l'invocation de scripts .rb et .rbw comme des commandes classiques (ce que nous n'utiliserons pas).

Autre point à régler sur Windows XP SP2 : l'avertissement du système chaque fois qu'on voudra lancer un serveur. La première fois que ça vous arrivera, choisissez Débloquer.

Sous Linux/BSD

La plupart des distributions Linux et des BSD ont un système de paquetages, avec Ruby et les outils connexes (notamment Rake) disponibles sous forme de paquets. Entre autres distributions proposant Ruby, on trouve :

- Debian et dérivés (Ubuntu, Kubuntu, etc.) ;
- Mandriva (anciennement Mandrake) ;
- SuSE et dérivés (comme OpenSuSE) ;
- Red Hat Linux (et Fedora) ;
- Slackware (au travers de paquets préparés sur LinuxPackages.net, par exemple) ;
- FreeBSD, NetBSD et OpenBSD.

Qui plus est, la majorité des distributions Linux ont déjà Ruby installé, car un nombre croissant d'outils Linux sont réalisés en Ruby. Commencez donc par ouvrir un shell et taper la commande :

```
| ruby -v
```

Si vous obtenez un numéro de version d'au moins 1.8 (par exemple « ruby 1.8.4 (2005-12-24) [i486-linux] ») plutôt qu'un message d'erreur du type « commande introuvable », vous n'avez aucune installation à faire. Dans le second cas, retentez tout de même votre chance avec la commande `ruby1.8` plutôt que simplement `ruby`.

Notez que pour disposer de WEBrick, la bibliothèque de serveur HTTP léger, il vous faut au moins la version 1.8.0 de Ruby, mais cette version datant d'août 2003, une distribution n'en disposant pas ferait véritablement figure de brontosaure.

Les utilisateurs de Linux n'ont normalement pas de difficulté à installer des paquets, qu'ils passent par des outils graphiques comme SynaptiK ou utilisent des outils en ligne de commande, par exemple `apt-get` ou `aptitude`. Le nom du paquet est en général tout simplement `ruby`.

Sous Mac OS X

Un installateur « 1 clic » pour Mac OS X est en cours de création à l'heure où j'écris ces lignes, par la même équipe que celle de son homologue Windows.

Toutefois, Ruby est d'ores et déjà simple à installer sous Mac OS X. Après tout, un très grand nombre de développeurs Ruby, et la majorité des développeurs Rails sont sur Mac OS X.

Tout d'abord, si vous tournez sous Jaguar (10.3) ou ultérieur, Ruby est installé d'entrée de jeu. Si vous êtes encore sur Panther (10.2), vous trouverez un `.dmg` de Ruby 1.8.2 sur la page <http://homepage.mac.com/discord/Ruby/> (d'ailleurs, il y en a un pour Jaguar aussi).

Depuis Puma (10.4), Ruby est très bien pris en charge sur Mac, au point que même Ruby on Rails est installé d'office !

Un mot sur le cache

Enfin, je précise que quel que soit le système d'exploitation, le cache du navigateur peut parfois empêcher le bon fonctionnement des exemples, en particulier lorsqu'on les teste les uns après les autres à un bref intervalle (les fichiers CSS et JS pour la plupart s'appellent `client.css` et `client.js`, mais ils changent d'un exemple à l'autre). L'effet de cookies mis à jour n'est pas non plus visible si on passe par le cache. L'annexe D traite en détail de la configuration du cache dans les principaux navigateurs. Effectuez cette manipulation, sous peine d'obtenir des comportements très curieux au fil des exemples...

Un petit serveur HTTP et un code dynamique simple

Voici un script Ruby qui contient tout notre premier serveur de test !

Listing 6-1 Notre premier serveur, avec deux actions distinctes

```
#! /usr/bin/env ruby ①

require 'webrick' ②
include WEBrick

server = HTTPServer.new(:Port => 8042) ③
server.mount_proc('/hi') do |request, response| ④
    response.body = 'Bonjour !'
end

server.mount_proc('/time') do |request, response| ⑤
    response.body = Time.now.to_s
end

trap('INT') { server.shutdown } ⑥

server.start ⑦
```

- ① (Linux/Mac OS X) Permet de lancer ce script comme un exécutable normal.
- ② Déclare un besoin du module webrick et l'importe dans notre script.
- ③ Configure un serveur HTTP sur le port 8042.
- ④ Associe l'URL /hi à une réponse figée, « Bonjour ! ».
- ⑤ Associe l'URL /time à une réponse dynamique, contenant la date et l'heure au moment de la requête.
- ⑥ Réagira à l'interruption (Ctrl+C) en arrêtant le serveur.
- ⑦ Démarré le serveur web !

Comme vous pouvez le constater, il ne faut que très peu de lignes, assez compréhensibles, pour créer de toutes pièces un serveur web, associer des contenus fixe ou dynamique à deux URL, s'assurer de pouvoir le fermer proprement, et finalement le démarrer !

Sauvegardez ce code dans un fichier nommé `hi_timed.rb`, puis ouvrez une console (ce que Windows nomme une « invite de commandes »), placez-vous dans le répertoire contenant le fichier, et tapez simplement :

```
| ruby hi_timed.rb
```

Vous allez obtenir un affichage similaire à ceci :

```
[2006-06-12 23:02:25] INFO WEBrick 1.3.1
[2006-06-12 23:02:25] INFO ruby 1.8.4 (2005-12-24) [i486-linux]
[2006-06-12 23:02:25] INFO WEBrick::HTTPServer#start: pid=16951
  ↬ port=8042
```

Ouvrez à présent un navigateur et allez sur `http://localhost:8042/hi`. Vous devez voir apparaître le texte « Bonjour ! ». Dans la console où vous avez lancé le serveur, vous remarquez qu'un journal de requêtes est en train de se constituer, ce qui est bien pratique lors du débogage :

```
[2006-06-12 23:02:25] INFO WEBrick 1.3.1
[2006-06-12 23:02:25] INFO ruby 1.8.4 (2005-12-24) [i486-linux]
[2006-06-12 23:02:25] INFO WEBrick::HTTPServer#start: pid=16951
  ↬ port=8042
localhost.localdomain - - [12/Jun/2006:23:04:11 CEST] "GET /hi
  ↬ HTTP/1.1" 200 9
- -> /hi
[2006-06-12 23:04:12] ERROR `/favicon.ico' not found.
localhost.localdomain - - [12/Jun/2006:23:04:12 CEST] "GET /favicon.ico
  ↬ HTTP/1.1" 404 281
- -> /favicon.ico
```

Chaque requête génère au moins deux lignes : la première qui identifie la machine cliente (ici, vous-même), la date et l'heure de la requête, le type de requête effectuée (par exemple « GET /hi HTTP/1.1 »), le code de réponse (200 : tout va bien) et la taille du contenu renvoyé (9 octets, soit le nombre de caractères dans « bonjour ! »). La seconde ligne indique le chemin absolu, au sein du serveur, de la ressource demandée.

Vous êtes peut-être surpris de voir une demande pour `/favicon.ico`, que vous n'avez pas faite. Tout navigateur la fait après une première requête sur un site, pour tenter de récupérer une petite icône identifiant le site, qui sera alors affichée à côté de l'URL, associée à un éventuel marque-page, placée dans l'onglet de la page s'il existe, etc.

Intéressons-nous à présent à l'action dynamique que nous avons mise en place, pour l'horodatage : naviguez sur `http://localhost:8042/time`. Vous obtenez un résultat du type :

```
Mon Jun 12 23:09:44 CEST 2006
```

Il s'agit de la représentation textuelle par défaut d'une date en Ruby, qui suit le format standard de la RFC 2822 (format qu'on retrouve, en interne, dans les protocoles de messagerie électronique et de consultation de pages web, par exemple). Ce n'est certes pas ce qu'il y a de plus lisible, mais qu'importe : cela nous permet de montrer l'aspect dynamique. En effet, si vous rafraîchissez la page dans votre navigateur, vous voyez

que la date est mise à jour. Le petit morceau de code Ruby dans notre serveur est invoqué à chaque requête et change son résultat à chaque fois. Nous avons là de quoi tester Ajax sur des bases plus intéressantes qu'avec un simple contenu statique.

Arrêtons à présent notre petit serveur d'exemple : il nous suffit de reprendre la console et de taper Ctrl+C. Le serveur s'arrête en clôturant son journal de quelques lignes stoïques :

```
[2006-06-12 23:13:35] INFO going to shutdown ...
[2006-06-12 23:13:35] INFO WEBrick::HTTPServer#start done.
```

À présent que nous avons tout le nécessaire pour simuler une application côté serveur, rendant ainsi notre exploration d'Ajax plus intéressante, il nous faut faire connaissance avec le véritable moteur d'Ajax, l'objet responsable du « A » initial : le mécanisme de requêtes asynchrones.

La petite histoire de XMLHttpRequest

C'est grâce à l'objet XMLHttpRequest qu'une page web, par l'intermédiaire de code JavaScript, peut « discuter », en coulisses, avec des serveurs (très généralement celui qui a fourni la page web ; des mécanismes portables et sécurisés pour interagir avec d'autres serveurs sont en cours de spécification, mais ça prend du temps...).

Origines et historique

Rendons à César ce qui lui appartient : dans une sphère technologique où Microsoft, depuis la sortie de MSIE 5 en 1999, n'a guère brillé par son sens de l'innovation, il faut souligner que c'est au sein de MSIE, et dès la version 5 justement, qu'est apparu XMLHttpRequest. Il était (tout comme dans les versions 5.5 et 6.0) fourni sous forme d'un ActiveX. La première implémentation compatible est apparue en 2002 dans Mozilla 1.0 (on le trouve donc également dans Firefox et Camino) et les autres navigateurs ont suivi le mouvement : Safari, Konqueror, Opera...

Au cœur d'Ajax, XMLHttpRequest est tellement utile et répandu que son API, à l'origine propriétaire, est en train de faire l'objet d'une standardisation par le W3C (<http://www.w3.org/TR/XMLHttpRequest/> ; une traduction française est disponible sur <http://www.xul.fr/XMLHttpRequest.html>) sur la base du dernier jet à l'heure actuelle (le septième, qui devrait en théorie être le dernier avant la spécification finalisée), en date du 15 avril 2008.

Bien préparer un échange asynchrone

D'un didacticiel à l'autre, on voit plusieurs manières de préparer une requête asynchrone. Comme nous l'avons vu en début de chapitre, le processus comporte un certain nombre d'étapes. Nous allons voir ensemble le détail, en insistant sur une méthodologie qui nous semble optimale. Mais avant, voici un petit rappel de la différence significative entre MSIE 6 et versions antérieures d'une part, et les autres navigateurs d'autre part.

ActiveX versus objet natif JavaScript

Jusqu'à la création d'une implémentation compatible par Mozilla en 2002, XMLHttpRequest restait une technologie centrée sur MSIE. Il n'est donc pas surprenant qu'il ait été fourni sous forme d'ActiveX, technologie qui, en 1999, n'avait pas encore vu sa réputation détruite à coups d'innombrables failles de sécurité et usages abusifs.

Cependant, les autres navigateurs, qui n'implémentent intentionnellement pas ActiveX, principalement pour des raisons de sécurité justement, ont naturellement opté pour un choix plus rationnel et, techniquement, plus simple : la mise à disposition de cette fonctionnalité sous forme d'objet natif JavaScript, bien plus facile à utiliser.

Cette forme de mise à disposition, la seule qui puisse être multinavigateur, est aujourd'hui la plus répandue ; elle est exigée par la standardisation en cours du W3C, et Microsoft a basculé vers ce mode de fourniture avec MSIE 7, il reste un schisme entre les deux modes de mise à disposition, fossé qui n'est pas près de se résorber : on sait déjà que MSIE 7 ne sera disponible que sur des Windows XP SP2 à la licence vérifiée, ce qui constitue une partie plutôt minoritaire du parc Windows déployé. Les entreprises, notamment, sont souvent restées sur Windows 2000, voire Windows Me ou Windows98. Pour tous ces postes, ainsi que les XP, XP SP1 et XP SP2 illégaux (très répandus, par exemple, sur le marché asiatique), il faudra choisir entre rester sur MSIE 6 ou passer sur un autre navigateur, la principale alternative étant Firefox utiliserait cette forme également : <http://blogs.msdn.com/ie/archive/2006/01/23/516393.aspx>. Utiliser XMLHttpRequest comme objet natif JavaScript est donc aujourd'hui possible sur plus de 70 % des navigateurs déployés, mais MSIE 6 reste la principale raison de devoir gérer le cas ActiveX...

Créer l'objet requêteur

Voici, ci-après, une première version, peut-être un peu brutale, d'une fonction portable de création d'un objet XMLHttpRequest (sans utiliser Prototype pour l'instant).

Listing 6-2 Une première tentative d'obtention portable

```
function getRequester() {
    try {
        return new ActiveXObject('Msxml2.XMLHTTP');
    } catch (e) {
    }
    try {
        return new ActiveXObject('Microsoft.XMLHTTP');
    } catch (e) {
    }
    try {
        return new XMLHttpRequest();
    } catch (e) {
    }
    return false;
}
```

On voit que l'objet a, au fil du temps, été disponible sous plusieurs noms différents : d'abord `Microsoft.XMLHTTP`, puis plus tard, `Msxml2.XMLHTTP`. Quant à son utilisation comme objet natif JavaScript, elle est des plus simples.

Cet algorithme a l'avantage de se factoriser facilement (le code Prototype qui réalise cette tâche, à savoir `Ajax.getTransport()`, n'est pas sans une certaine esthétique), mais si vous souhaitez minimiser le nombre d'exceptions levées, vous pouvez utiliser une autre variante, très répandue dans les articles et didacticiels sur le sujet.

Listing 6-3 Une version avec les commentaires conditionnels MSIE

```
function getRequester() {
    var result = false;
/*@cc_on */
/*@if @_jscript_version >= 5)
    try {
        result = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
        try {
            result = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (E) {
            result = false;
        }
    }
@end */
}
```

```
if (!result && 'undefined' != typeof XMLHttpRequest) {  
    try {  
        result = new XMLHttpRequest();  
    } catch (e) {  
        result = false;  
    }  
}  
return result;  
}
```

Notez les syntaxes spécifiques à MSIE, qui constituent des commentaires conditionnels. Ici, MSIE ignorera le commentaire s'il est sur une version inférieure à 5, tandis que les autres navigateurs les ignoreront systématiquement.

Ce code peut sembler futé, mais à mon sens, il s'agit là de beaucoup de bruit pour rien, sans parler de la pollution visuelle engendrée par la syntaxe propriétaire des commentaires conditionnels. Dans une application web, même exigeante, on imagine difficilement plus d'une dizaine de demandes de requêteurs à la seconde (et le plus souvent, bien moins que ça), et le « gaspillage » de temps causé par les levées supplémentaires d'exceptions se chiffre au pire en centièmes de secondes.

C'est pourquoi, à l'issue de cette section consacrée à examiner les entrailles de l'API XMLHttpRequest, nous utiliserons les enrobages de confort proposés par Prototype ; même si, par exemple, l'obtention d'un requêteur y utilise un algorithme similaire à notre première tentative, le gain de productivité et l'écart infinitésimal des performances justifient pleinement cette décision.

Décrire notre requête

Techniquement, l'ordre recommandé d'initialisation d'un objet XMLHttpRequest suppose une étape supplémentaire avant de préparer la requête ; pourtant, nous allons suivre un ordre plus logique pour l'instant et céder aux exigences techniques un peu plus tard.

Pour configurer une requête, on doit d'abord l'ouvrir (et ce n'est pas tous les jours qu'on vous encourage à l'ouvrir), en indiquant :

- 1 la méthode HTTP utilisée (par exemple, GET ou POST) ;
- 2 l'URL de la ressource dynamique à requêter côté serveur ;
- 3 le mode de synchronisation (asynchrone par défaut, mais on peut passer en synchrone, bien que ce soit plutôt une hérésie !) ;
- 4 une authentification HTTP optionnelle (identifiant, mot de passe), si la ressource côté serveur en nécessite une.

Cette initialisation s'effectue avec la méthode `open`. Par exemple :

```
var requester = getRequester();
requester.open("POST", "/blog/comment", true);
```

Une requête HTTP, outre sa méthode et son URL cible, est constituée (comme nombre de requêtes/réponses dans les protocoles qui font vivre le Web) d'en-têtes et d'un corps. Les en-têtes permettent de décrire une foule de choses, comme le type de résultat attendu, les formats de données acceptés, le type d'encodage employé pour le corps de la requête, etc. On les définit à l'aide de la méthode `setRequestHeader`.

Ainsi par exemple, il est parfois nécessaire de circonvenir la politique agressive de cache des résultats de requêtes GET mise en œuvre par MSIE, notamment dans le cadre de requêtes internes effectuées à intervalles potentiellement très faibles. On peut réaliser cela en utilisant un en-tête approprié pour la requête, comme ceci :

```
requester.setRequestHeader("If-Modified-Since", "Sat, 1 Jan 2000
00:00:00 GMT");
```

Quant au corps de la requête, il faut d'abord savoir qu'il n'y en a pas forcément. Ainsi, une requête de type GET encode tous ses paramètres dans l'URL, ce qui suppose un envoi de corps vide (`null`). En revanche, une requête POST ou PUT utilise fréquemment des données fournies par le corps de la requête (les champs d'un formulaire y figurent, par exemple).

Envoyer la requête

Le corps de la requête est fourni au moment de l'**envoi** de celle-ci, à l'aide de la méthode `send`. Voici deux exemples de requête, une en GET et une en POST :

Listing 6-4 Deux exemples de requête, l'une en GET, l'autre en POST

```
requester.open("GET", "/blog/comments?article_id=183");
requester.send(null);
...
requester.open("POST", "/blog/comment");
requester.send("article_id=183&author_name=TDD&author_email=tdd@example
.com&comment=Voici+un+exemple+de+requete+POST");
```

Ne soyez pas alarmés par l'aspect du deuxième appel à `send` : cet encodage particulier des données, qui correspond à l'encodage par défaut pour les méthodes POST et PUT,

à savoir le format `application/x-www-form-urlencoded`, peut être réalisé automatiquement par JavaScript.

Une fois la requête lancée, il ne reste plus qu'à en suivre la progression pour finalement récupérer les résultats éventuels.

Recevoir et traiter la réponse

C'est ici que la réalité technique nous rattrape, car pour suivre la progression de la requête (de sa création à la fin de la récupération de la réponse du serveur), il faut s'y être pris à l'avance, en créant ce qu'on appelle une fonction de rappel, c'est-à-dire une fonction de votre cru, respectant une certaine forme, qui sera appelée par le système dans certaines circonstances.

Ici, il s'agit d'une fonction sans argument particulier, mais qui devra pouvoir accéder à votre objet requêteur pour interroger son état. On l'associe à l'objet en l'affectant à la propriété `onreadystatechange`. Cette association se fait de préférence avant l'appel à `open` (sauf si on tente de réutiliser le même requêteur par la suite dans MSIE, auquel cas il vaut mieux la faire après).

La fonction va interroger l'état actuel du requêteur, en examinant sa propriété `readyState`. Au fur et à mesure du cycle de vie du requêteur, cette propriété peut prendre les valeurs suivantes :

- 1 non initialisé (pas encore d'appel à `open`) ;
- 2 ouvert (un appel à `open` a été correctement exécuté) ;
- 3 envoyé (la requête a été préparée, l'appel à `send` a eu lieu) ;
- 4 en réception (des données arrivent, mais ce n'est pas fini) ;
- 5 réception terminée (toutes les données sont arrivées, on peut à présent consulter la réponse du serveur).

Voici un code mettant en place une fonction de rappel qui, lorsque la requête aura abouti, affichera le résultat dans une boîte de message :

```
requester.onreadystatechange = function() {
    if (4 == requester.readyState && 200 == requester.status)
        alert(requester.responseText);
};
```

Le résultat de la requête comprend un statut HTTP (le code 200 indique que tout s'est bien passé), consultable *via* la propriété `status`, et un corps de réponse. La propriété `responseText` fournit le « texte brut » de cette réponse, quel qu'en soit le

format (XML, JSON, XHTML, etc.), mais si la réponse est un format XML et qu'on souhaite le traiter comme tel (comme un document XML manipulable à l'aide des interfaces du DOM), on peut utiliser `responseXML`.

Une utilisation complète de notre petit serveur d'exemple

Nous allons à présent assembler ces fragments de code dans un tout cohérent. Pour commencer, il va nous falloir quelques fichiers statiques : une page HTML mais aussi un fichier JavaScript (et même deux, car nous allons utiliser Prototype pour nous simplifier les parties non Ajax).

Le script serveur que nous avions écrit ne prévoit pas de laisser un accès à un répertoire du disque : seuls deux points d'accès sont prévus, `/hi` et `/time`, qui amènent tout droit sur du code Ruby. Nous devons donc arrêter ce script (un simple `Ctrl+C` dans la console), le modifier comme ci-après, et le relancer (en tapant simplement `ruby hi_timed.rb`, comme tout à l'heure) après avoir créé, au même niveau, un répertoire `docroot` destiné à accueillir nos fichiers statiques.

Listing 6-5 Script serveur modifié pour gérer un répertoire racine

```
#!/usr/bin/env ruby

require 'webrick'
include WEBrick

server = HTTPServer.new(:Port => 8042)

server.mount('/', HTTPServlet::FileHandler, './docroot') ①

server.mount_proc('/hi') do |request, response|
  response.body = 'Bonjour !'
end

server.mount_proc('/time') do |request, response|
  response.body = Time.now.to_s
end

trap('INT') { server.shutdown }

server.start
```

La ligne ① associe à la racine de nos URL le répertoire `docroot` situé au même niveau que notre script `hi_timed.rb`.

Le nom du répertoire n'a bien sûr rien d'obligatoire, puisque nous le précisons dans le script. Nous allons y déposer trois fichiers :

- 1 Notre fichier HTML, qui fournit simplement la page de test.
- 2 Le fichier JavaScript de Prototype.
- 3 Notre fichier JavaScript, qui va associer des comportements aux boutons de notre page, et effectuer les requêtes Ajax.

Voici le code, très simple, de notre fichier HTML.

Listing 6-6 Notre page hi_timed_client.html pour cet exemple trivial d'Ajax

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
  ↪ charset=iso-8859-15" />
  <title>Exemple Ajax trivial</title>
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="hi_timed_client.js"></script>
</head>
<body>

<h1>Exemple Ajax trivial</h1>

<form>
<p>
  <input type="button" id="btnGreet" value="Salut !" />
  <input type="button" id="btnWhatTime" value="Quelle heure est-il ?" />
</p>
</form>

</body>
</html>
```

Vous pouvez récupérer le fichier JavaScript de Prototype dans l'archive des codes source de ce livre, sur le site des éditions Eyrolles.

Enfin, voici le code source de notre fichier JavaScript, avec quelques commentaires de rappel sur la partie Ajax. Si vous avez des doutes sur les autres parties (association de gestionnaires d'événements, etc.), feuilletuez donc les chapitres 3 et 4 pour retrouver vos marques.

Listing 6-7 Notre fichier JavaScript, `hi_timed_client.js`

```
function askTime(e) { ①
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status)
            alert('Maintenant : ' + requester.responseText);
    };
    requester.open('GET', '/time', true);
    requester.send(null);
} // askTime

function bindButtons(e) {
    $('btnGreet').observe('click', greet);
    $('btnWhatTime').observe('click', askTime);
} // bindButtons

function getRequester() { ②
    var result = false;
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
    try {
        result = new ActiveXObject("Msxml2.XMLHTTP");
    } catch (e) {
        try {
            result = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (E) {
            result = false;
        }
    }
@end @_*/
    if (!result && 'undefined' != typeof XMLHttpRequest) {
        try {
            result = new XMLHttpRequest();
        } catch (e) {
            result = false;
        }
    }
    return result;
} // getRequester

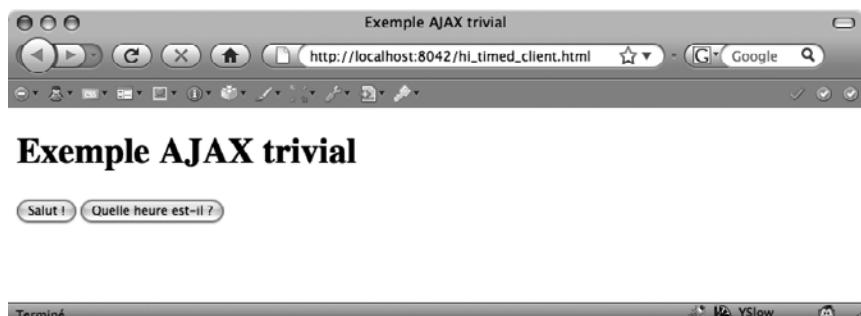
function greet(e) { ③
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status)
            alert(requester.responseText);
    };
    requester.open('GET', '/hi', true);
    requester.send(null);
} // greet

document.observe('dom:loaded', bindButtons);
```

- ➊ Voici la fonction qu'on associera à notre bouton « Quelle heure est-il ? ». On y retrouve les différents blocs de code vus plus haut : l'obtention d'un requêteur, l'association d'un gestionnaire d'état, la détection d'une réponse terminée et valide ainsi que l'envoi de la requête.
- ➋ Revoici notre fonction d'obtention de requêteur, avec ses commentaires conditionnels MSIE. On pourrait polémiquer longuement sur le meilleur algorithme d'obtention, mais ce serait inutile puisque, à partir du prochain chapitre, nous basculerons sur les fonctions Ajax de Prototype de toutes façons.
- ➌ Voici le gestionnaire associé au clic sur le bouton « Salut ! ». Il est presque identique à celui de « Quelle heure est-il ? ». La seule différence est qu'il ne préfixe pas le texte de retour.

Voilà, les acteurs sont en place, la première peut commencer : si ce n'est déjà fait, lancez votre script serveur, puis ouvrez un navigateur et allez sur http://localhost:8042/hi_timed_client.html. Vous devriez obtenir un affichage similaire à celui-ci.

Figure 6–1
Notre page de test



Cliquez sur le bouton « Salut ! ». La page ne se recharge absolument pas, mais un petit appel du pied au serveur nous renvoie le texte « Bonjour ! », que notre gestionnaire d'état s'empresse d'afficher avec un `alert`.

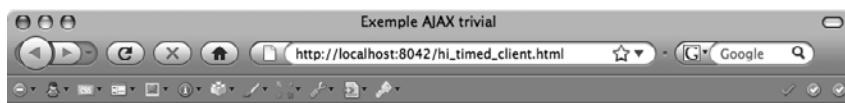
Essayez à présent avec le bouton « Quelle heure est-il ? » : vous obtenez un affichage (certes en anglais, et plus précisément au format défini par la RFC 2822) de la date et l'heure courante. Plusieurs tentatives amènent évidemment un résultat différent, car le temps passe...

Comment surveiller les échanges Ajax de nos pages ?

Lorsqu'on met au point une fonctionnalité basée sur Ajax et que des bogues apparaissent, il est vite frustrant de ne pas avoir un œil sur le contenu détaillé des réponses que nous fournit le serveur.

Nous avons déjà exploré l'essentiel de l'extension Firebug et ses inspecteurs et débogeurs pour le DOM, CSS, JavaScript et HTML, tous facilement visibles dans un espace en bas de page. Cette extension permet aussi de surveiller les requêtes et réponses récupérées par XMLHttpRequest.

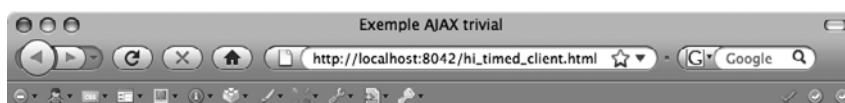
Figure 6–2
Le panneau Firebug
en plein travail



Exemple AJAX trivial

Pour activer le suivi des requêtes Ajax (désactivé par défaut), déroulez le menu Options du panneau de gauche et cochez l'option Show XMLHttpRequests. Après quoi, voici l'aspect du panneau après un clic sur chacun de nos deux boutons :

Figure 6–3
Firebug nous signale
nos deux requêtes Ajax.

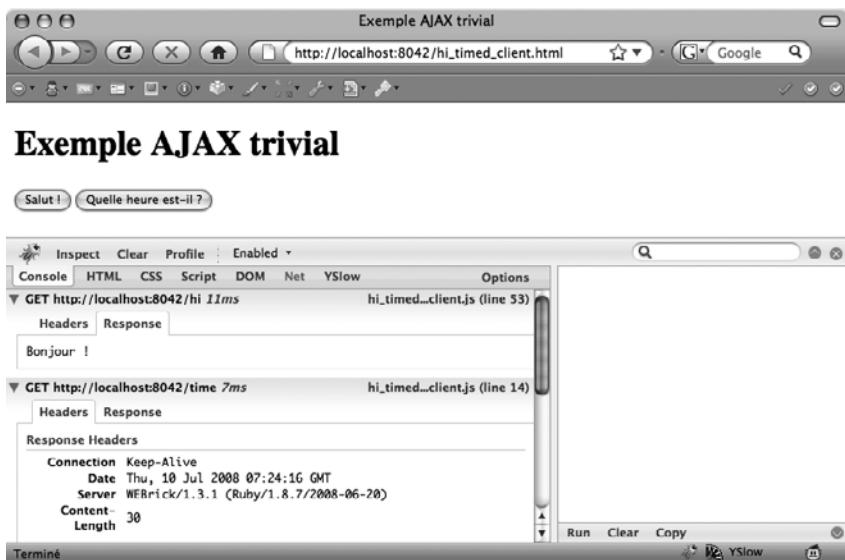


Exemple AJAX trivial

En déroulant ces lignes de résultat et en choisissant l'onglet Headers pour l'un d'eux, on a une idée des informations (détailées !) qui nous sont proposées :

Figure 6–4

Tant les en-têtes que le corps de réponse sont disponibles.



Il existe de nombreux outils d'aide au développement web, et même si à l'heure où j'écris ces lignes, MSIE ne dispose pas, à ma connaissance, d'une aide au suivi des requêtes Ajax, il existera peut-être une solution, même partielle, d'ici la parution de ce livre.

Types de réponse : XHTML, XML, JS, JSON...

Les lecteurs attentifs (je suis sûr que vous en faites partie) auront sans doute remarqué dans l'avant-propos une mention de réponses Ajax utilisant autre chose que du XHTML ou du XML. Après tout, si nous examinons les réponses de notre petit exemple précédent, nous voyons qu'il ne renvoie pas un format particulier : il renvoie une seule donnée, textuelle, très simple. Pas de balisage XML, pas de balises XHTML, pas de code JavaScript...

En effet, dans la mesure où c'est votre code qui interprète la réponse, vous avez le champ libre quant au format de celle-ci. Il vous appartient de choisir au mieux, en utilisant un format qui facilite le traitement côté client, sans pour autant être pénible à générer côté serveur.

Il existe plusieurs types de réponse assez répandus, que nous allons exposer un peu plus loin, avec un exemple concret pour chacun. Mais gardez à l'esprit qu'il n'y a pas vraiment de contraintes, hormis la difficulté que poserait probablement le traitement

de données binaires en JavaScript : on se limitera normalement à des réponses de type texte.

Bien choisir son type de réponse

Il s'agit de réfléchir au cas par cas. Dans l'exemple précédent, nous souhaitions simplement récupérer une information unique, pour ne pas dire atomique, sans structure ou complexité aucune. Ne pas s'encombrer d'un format spécifique est alors une bonne solution ! D'autres fois, il faudra choisir judicieusement le format. Comment faire ?

Commençons déjà par nous poser la question de ce qui constitue un « bon candidat ». Un bon format pour une réponse Ajax devrait, idéalement, satisfaire aux critères suivants :

- Il est facile à traiter côté client, en JavaScript donc. On verra qu'à l'heure actuelle, cela limite (nativement en tout cas) les réponses XML vers MSIE et Safari. En revanche, du XHTML généré côté serveur peut être facilement injecté dans le DOM de la page, du JavaScript peut être évalué, ainsi que JSON, fatallement (pourquoi fatallement ? Un peu de patience...).
- Il n'est pas inutilement encombrant : plus la représentation est simple, plus son analyse est facile, notamment pour un format structuré qui n'est pas couvert par les objets JavaScript disponibles.
- Il est facile à générer côté serveur. Ce dernier point est généralement garanti : à moins d'utiliser une technologie quelque peu contraignante côté serveur (je ne peux m'empêcher de penser aux pages ASP en Visual Basic), celui-ci dispose de toute la richesse fonctionnelle nécessaire à la création de tous types de contenus.

Sur ces bases, nous pouvons passer en revue les différents formats de réponse courants :

- **Texte simple** : par définition, trivial à générer comme à traiter côté client. Particulièrement approprié pour les données toutes simples : petits morceaux de texte, valeurs numériques (pouvant représenter aussi des dates et heures, des booléens et bien d'autres choses), etc.
- **XHTML** : côté client, c'est trivial. On peut insérer le fragment retourné au beau milieu du DOM de la page, en douceur. Ce n'est pourtant pas toujours approprié, techniquement ou conceptuellement, comme nous le détaillerons plus loin.
- **XML** : relativement facile à générer côté serveur, sa facilité de traitement côté client dépend directement de sa taille et du navigateur. Certains navigateurs ne prennent pas en charge le DOM niveau 3 XPath, ce qui rend tout de suite plus compliquée l'analyse de XML complexe. Ce n'est en revanche pas un souci pour des documents de faible envergure. C'est d'ailleurs un des deux formats les plus appropriés pour des données structurées.

- **JavaScript** : assez facile à générer côté serveur, il est absolument trivial à traiter côté client : il suffit de l'évaluer, à l'aide d'un simple appel de méthode. Évidemment, il faut être sûr de la fiabilité du script ainsi récupéré. On limite en général à ses propres serveurs... Parfait pour permettre à la couche serveur de renvoyer des traitements de nature dynamique (on verra des exemples plus loin).
- **JSON** : facile à générer côté serveur et trivial à traiter côté client, puisqu'il s'agit en fait d'une expression JavaScript valide. Idéal pour passer des informations structurées spécifiques, le code de traitement JavaScript pouvant être assez réduit par comparaison à l'analyse d'une grappe XML équivalente.

Mais trêve de conseils généraux, rien ne vaut la pratique ! Nous allons mettre en œuvre un exemple concret pour chaque cas de figure, afin de vous aider à mieux cerner les avantages et inconvénients de chaque format. Ce sera aussi l'occasion de détailler leurs contextes techniques.

Gardez à l'esprit que nombre de ces exemples seraient probablement plus simples en utilisant une bibliothèque comme Prototype. C'est d'ailleurs ce que nous ferons, sur des données plus riches, au chapitre suivant. Mais il est important que vous saisissez bien, au préalable, les tenants et aboutissants de chaque démarche.

Une réponse textuelle simple : renvoyer une donnée basique

Ce type de réponse est approprié à bien des cas de figure. Afin d'ancrer la notion, nous allons voir deux exemples : le premier utilisant des cookies mais simple côté client, le second très simple côté serveur, mais un peu plus lourd côté client.

Exemple 1 : sauvegarde automatique

Dans une application Ajax, il arrive fréquemment que des requêtes Ajax n'attendent pas de résultat particulier. De telles requêtes servent généralement à « tenir le serveur au courant » de ce qui se passe côté client, par exemple de la personnalisation de l'interface qu'effectue l'utilisateur (comme le changement de la disposition des blocs de contenu), l'ajout de contenu, etc. Lorsque l'information est créée côté client et que le serveur n'a rien à y apporter, il se contente de l'enregistrer.

En revanche, la partie client aimeraient sans doute savoir si l'enregistrement s'est bien passé. Peut-être la couche serveur a-t-elle expiré sa session, nécessitant une nouvelle authentification ? Peut-être a-t-elle un souci d'accès à la base de données, ou un problème de quota disque ? Ces problèmes n'engendreront pas forcément une erreur de la couche serveur (que nous pourrions détecter en examinant la propriété `status`). Afin de construire un système robuste, nous devons développer un code paré à toute éventualité, et donc à même de savoir si le traitement n'a pu s'effectuer.

En termes de requête/réponse, c'est assez simple : il suffit de définir que ces requêtes attendent une réponse exprimant un code de statut. On peut s'inspirer des codes HTTP, et décider par exemple que nous renverrons une ligne de texte démarrant par un code numérique générique (200 = OK, etc.), suivi d'un éventuel message à usage plutôt interne, potentiellement utile au débogage.

Afin d'illustrer un tel comportement, nous allons réaliser un petit système de sauvegarde automatique d'un champ texte. Le scénario d'utilisation de la page est le suivant :

- 1 On arrive sur la page pour la première fois. Le champ a une valeur par défaut, par exemple « Saisissez votre nom ici ».
- 2 On modifie le nom et on quitte le champ en cliquant ailleurs sur la page, en cliquant sur un lien, en changeant d'onglet... Tout, sauf fermer directement la page (et encore, une seule ligne de script en plus gérerait ce cas si nous le jugeons pertinent).
- 3 Lorsqu'on revient sur la page, le champ a une valeur à jour.

Pour réaliser cela, trois conditions doivent être satisfaites :

- Le serveur doit conserver une information associée au côté client. On va passer par des cookies pour identifier le client et maintenir un tableau associatif côté serveur.
- Le contenu du champ doit être généré dynamiquement. On pourrait se contenter de demander à la couche serveur sa valeur une fois la page chargée, mais cela entraînerait une latence entre l'affichage de la page et la mise à jour de la valeur du champ, qui serait potentiellement désagréable. Il faut donc que la page entière soit générée.
- Le navigateur doit réagir lorsque le champ de saisie perd le focus (cessé d'être la cible de la saisie clavier) en synchronisant la valeur sur le serveur ; si cette synchronisation échoue, il doit le signaler à l'utilisateur.

Afin de générer du contenu côté client, nous allons stocker le modèle de la page dans un fichier HTML, et utiliser une syntaxe spéciale pour y placer des portions dynamiques. On utilisera la bibliothèque ERb, fournie avec Ruby, dont la syntaxe ressemble beaucoup à ASP ou JSP.

Commencez par arrêter, si ce n'est déjà fait, notre serveur de test précédent. Créez un nouveau répertoire de travail pour cet exemple, appelons-le `sauvegarde_auto`. Nous allons d'abord y déposer notre fichier modèle, `modele.rhtml` (car `.rhtml` est l'extension conventionnelle des fichiers ERb).

Listing 6-8 Notre fichier de modèle pour la page d'exemple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  => xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
  => charset=iso-8859-15" />
  <title>Sauvegarde automatique</title>
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Sauvegarde automatique</h1>

<p>La saisie ci-dessous doit persister toute seule, si vous avez
JavaScript et
les cookies activés.</p>

<form>
<p><input type="text" name="name" id="edtName"
  value="<%= CGI::escapeHTML(name || '') %>" /></p>
</form>

</body>
</html>
```

La balise `form` n'est là que pour respecter la DTD de HTML. Notez l'attribut `value` du champ : `<%= CGI::escapeHTML(name || '') %>`. Il s'agit d'une syntaxe ERb, que ce dernier remplacera dynamiquement à l'exécution. On prend soin de « blinder » le contenu contre tout caractère ayant un sens en HTML, afin d'éviter les risques d'attaque par *injection*. Le fragment `|| ''` utilisera une chaîne vide par défaut si, pour des raisons de redémarrage du serveur, on utilisait au final un *cookie* sans correspondance serveur, aboutissant à un `name` inexistant.

Avant d'attaquer le code du serveur, on remarque toutefois que notre fichier aura besoin de deux scripts pour dialoguer avec le serveur : Prototype bien sûr, pour simplifier le script comme nous avons pris l'habitude de le faire, et un script dédié à notre exemple.

Créons donc un sous-répertoire `docroot` et plaçons-y `prototype.js`, plus un script `client.js` que nous pouvons dériver de `hi_timed_client.js` (pour récupérer la fonction `getRequester()`). Nous ajusterons ce second script un peu plus tard. Voici notre serveur, stocké dans un fichier `serveur.rb`, à la racine de notre répertoire de test, où se trouve aussi `modele.rhtml` :

Listing 6-9 Notre serveur pour cet exemple

```
#!/usr/bin/env ruby

require 'cgi'
require 'erb' ①
require 'webrick'
include WEBrick

DEFAULT_NAME = 'Saisissez votre nom ici'
names = {} ②
sessionGen = 0
template_text = File.read('modele.rhtml') ③
page = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/page') do |request, response| ④
  name = request.cookies.empty? ? DEFAULT_NAME :
    ↗ names[request.cookies[0].value.to_i]
  response['Content-Type'] = 'text/html'
  response.body = page.result(binding)
end

server.mount_proc('/save_name') do |request, response|
  response['Content-Type'] = 'text/plain'
  if 0 == rand(4) ⑤
    response.body = '501 Could not be saved.'
    next
  end
  if request.cookies.empty? ⑥
    sessionGen += 1
    session_id = sessionGen
    cookie = Cookie.new('session_id', session_id.to_s)
    cookie.expires = Time.utc(2010, 12, 31)
    response.cookies << cookie
  else
    session_id = request.cookies[0].value.to_i
  end
  post = 'POST' == request.meta_vars['REQUEST_METHOD'] ⑦
  params = post ? CGI::parse(request.body) : request.query
  params.each { |k, v| params[k] = v[0] } if post
  names[session_id] = params['name']
  names[session_id] = request.query['name']
  response.body = '200 Saved.'
end

trap('INT') { server.shutdown }
```

```
srand  
server.start
```

- ❶ Nous avons besoin du module ERb pour interpréter la syntaxe de notre modèle HTML. Le module CGI nous permet d'analyser la requête POST et d'échapper (« blinder ») les affichages.
- ❷ Conteneur associant clefs de sessions et noms sauvegardés, et générateur de clefs.
- ❸ Chargement du texte du modèle depuis le fichier, et construction du moteur pour l'interpréter. L'extension .rhtml est juste une convention...
- ❹ Il faut fournir la variable `name`, qui contient soit la valeur par défaut, soit celle associée à la session. L'appel à `result` construit la page. `binding` représente la portée courante, dans laquelle ERb va chercher la variable `name`.
- ❺ Simulation d'un échec environ une fois sur quatre.
- ❻ Si c'est la première sauvegarde (pas de cookie existant), on crée une clef de session qu'on envoie dans un cookie valable jusqu'au 31/12/2012 (ça nous donne le temps de tester...). Ensuite on met à jour l'information transmise dans le conteneur de noms, côté serveur.
Ces quelques lignes permettent de traiter indifféremment du POST ou du GET. Afin de respecter les recommandations du W3C, nous allons utiliser un POST pour sauvegarder le nom sur le serveur, car il s'agit d'une opération modificatrice, pas d'une simple consultation.

Cet exemple ne tiendrait pas une forte charge, dans la mesure où le générateur de clefs de sessions n'est pas protégé contre des accès concurrents. Mais pour notre exemple, qu'importe !

Plus important : le choix du stockage de nom côté serveur, plutôt que d'utiliser directement le cookie. C'est une bonne habitude à prendre, principalement pour deux raisons :

- ❶ Un cookie n'a qu'une capacité limitée de stockage, pour des raisons de performances essentiellement. Il est acceptable d'y stocker un numéro (comme ici) ou un texte de faible longueur (clefs de session plus communes en ASP, PHP, JSP, Rails...). En revanche, stocker des contenus plus lourds va poser problème. Pour éviter deux poids deux mesures, on stocke donc nos données côté serveur.
- ❷ Un cookie est stocké en clair sur le poste client, ce qui peut constituer un souci si on venait à y stocker des données sensibles voire confidentielles. Certaines attaques XSS (*Cross-Site Scripting*, où un script est injecté sur un site pour transmettre ses cookies à un tiers) permettent à d'autres sites que le nôtre de récupérer des cookies qui ne leur appartiennent pas. Évitons donc d'y stocker des données « utiles ».

À présent que nous avons notre couche serveur au grand complet, nous pouvons la tester manuellement, pour ensuite passer à l'écriture du script qui synchronisera automatiquement côté serveur, en Ajax.

Lancez votre serveur depuis une ligne de commande placée sur le répertoire, en tapant simplement `ruby serveur.rb`.

Dans votre navigateur, allez sur l'URL `http://localhost:8042/page`.

Figure 6–5

Votre page au premier accès : le texte par défaut



Nous n'avons pas encore écrit le script qui va synchroniser notre saisie, alors réalisons une invocation à la main, en naviguant à présent sur l'URL suivante :

`http://localhost:8042/save_name?name=Christophe`.

(C'est pour cela que nous avons décidé de traiter le GET aussi : pour que vous puissiez tester comme ceci...) Vous devriez obtenir un message de code 200 ou 501. Vous pouvez rafraîchir la page plusieurs fois pour voir s'afficher l'un ou l'autre, suivant le résultat du « lancer de dé » côté serveur. Une fois que vous avez obtenu au moins un code 200, revenez sur votre page principale (`http://localhost:8042/page`) et, si nécessaire, rafraîchissez-la.

Figure 6–6

L'affichage initial prenant en compte notre sauvegarde



Vous pouvez examiner la valeur de vos cookies pour l'hôte `localhost`, afin de voir ce que le serveur vous a renvoyé lors de la sauvegarde initiale. Le mode d'emploi varie d'un navigateur à l'autre. Sur Firefox avec l'extension Web Developer, il suffit de cliquer `Cookies > View cookie information` dans la barre associée.

Figure 6-7

L'information stockée dans un cookie sur notre navigateur

The screenshot shows a web browser window titled "Cookie Information - http://localhost:8042/page". Below the title bar, there are two tabs: "Sauvegarde automatique" and "Cookie Information - http://loc...". The main content area displays the cookie information for the current page. A table titled "1 cookie" lists the following details:

NAME	session_id
VALUE	1
HOST	localhost
PATH	/
SECURE	No
EXPIRES	Fri, 31 Dec 2010 00:00:00 GMT

Below the table are two buttons: "Edit Cookie" and "Delete Cookie". At the bottom of the page, there is a "Terminé" button.

On voit bien notre cookie `session_id`, de valeur 1. L'information à proprement parler (le nom sauvegardé) n'existe pas dans le cookie : seul le serveur en dispose.

Il nous reste à écrire le script ! On va conserver la fonction `getRequester` de notre script de première démonstration, mais au lieu de ses fonctions `askTime`, `bindButtons` et `greet`, nous allons écrire le code suivant, et ajuster l'appel à `document.observe` en fin de script.

Listing 6-10 Notre script dédié, client.js

```
function bindTextField(e) { ①
    $('#edtName').observe('blur', syncName);
} // bindTextField

function getRequester() {
...
} // getRequester
function syncName(e) {
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState
            && (200 != requester.status ②
                || !/^200/.test( requester.responseText)))
            alert("Le nouveau nom n'a pas pu être sauvegardé !");
    };
}
```

```

var qs = $('edtName').serialize();
requester.open(POST, '/save_name', true);
requester.send(qs);
} // syncName

document.observe('dom:loaded', bindTextField);

```

- ➊ On associe notre fonction à l'événement `blur` du champ, c'est-à-dire la perte de focus. Toute saisie suivie d'une perte de curseur (clic en dehors de la zone de saisie, notamment suivi de lien, changement d'onglet, ouverture de menu, etc.) lancera la sauvegarde. En l'associant à l'événement `unload` de `window`, on pourrait aussi sauver sur fermeture directe.
- ➋ Un problème peut venir de HTTP (on teste donc `status`), mais aussi se situer à l'intérieur de notre couche serveur. En examinant le début du texte renvoyé, on détecte un code à problème.

Notez l'astuce qui consiste à utiliser la méthode `serialize` (fournie par Prototype) pour construire correctement la *query string* encodée que nous enverrons comme corps de données POST...

Et voilà, ça marche ! Essayez : rafraîchissez la page, modifiez le nom et cliquez n'importe où ailleurs (ou simplement changez d'onglet, allez sur une autre application, revenez...) et rafraîchissez : votre nouvelle valeur est bien là. Si vous avez installé Firebug, vous pouvez ouvrir son panneau pour suivre les requêtes de la page.

Figure 6–8

Modifications et clics en dehors de la zone de saisie, suivis par Firebug



Exemple 2 : barre de progression d'un traitement serveur

Autorisons-nous à présent un exemple un peu plus « sexy », mêlant Ajax et CSS pour obtenir quelque chose de plus joli qu'une simple sauvegarde automatique (laquelle, il faut bien l'avouer, recèle un potentiel esthétique plutôt léger). Nous allons réaliser une barre de progression dans une page web, qui va se mettre à jour toute seule pour suivre l'avancée d'un traitement sur le serveur. Sans Ajax, ce type de fonctionnalité exige des iframes, voire des cadres. Rien de tout cela pour nous : une seule page, sans cadre.

Bien entendu, nul besoin d'effectuer un véritable traitement côté serveur. Au risque de passer pour des fainéants, nous allons nous contenter de simuler un traitement : chaque fois qu'on nous demandera de signaler notre progression, nous nous contenterons d'augmenter un pourcentage, sans rien faire pour autant. Une fois en bout de course, on repart à zéro, histoire de pouvoir rafraîchir la page pour refaire la démonstration. En somme, on prototype, on maquette, mais on ne fait pas vraiment le boulot (c'est déjà mieux que de n'avoir qu'un Powerpoint à montrer).

Créez un nouveau répertoire de travail, avec comme d'habitude un sous-répertoire docroot contenant prototype.js. Vous pouvez aussi y recopier notre client.js récent, afin de récupérer getRequester.

Voici notre script serveur.rb, regardez comme il est petit !

Listing 6-11 Notre serveur simulant une progression de tâche

```
#!/usr/bin/env ruby

require 'webrick'
include WEBrick

progress = 0

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')
server.mount_proc('/whatsup') do |request, response|
  response['Content-Type'] = 'text/plain'
  progress += rand(5) + 1
  progress = 100 if progress > 100
  response.body = progress.to_s
  # Boucler pour la prochaine séquence d'appel ;-
  progress = 0 if 100 == progress
end

trap('INT') { server.shutdown }

srand
server.start
```

Rien de bien extraordinaire, comme vous pouvez le voir. On fait grimper le pourcentage de progression à chaque appel, d'un pas entre 1 et 5. Si on dépasse 100, on rabote la valeur. Et une fois arrivé au bout, on repart à zéro pour la démonstration suivante (voilà un code bien loin d'une implémentation réelle...).

Côté client, il va nous falloir une page web toute simple, sans partie dynamique, contenant notre barre de progression. Nous allons réaliser celle-ci entièrement en CSS, et histoire de sacrifier aux traditions (certes toutes fraîches), nous allons y coller l'incontournable *spinner*, cette petite animation omniprésente sur les sites Web 2.0, qui indique qu'on attend après Ajax. Notez que si vous optez ici pour la version la plus classique, vous êtes libre d'utiliser l'animation que vous voulez. Sans doute trouverez-vous votre bonheur sur cette petite collection en ligne : <http://www.napyfab.com/ajax-indicators/>.

Voici donc notre page. Comme elle est servie statiquement par notre serveur, nous allons simplifier l'URL en la nommant `index.html`, dans docroot.

Listing 6-12 La page web client et sa barre de progression CSS

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          charset=iso-8859-15" />
    <title>Barre de progression</title>
    <link rel="stylesheet" type="text/css" href="client.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>
<h1>Barre de progression</h1>

<p>Suivez le déroulement du processus côté serveur avec la barre ci-dessous.</p>

<div class="progressBar" id="progress">
    <span class="pbFrame">
        <span class="pbColorFill"></span>
        <span class="pbPercentage">0%</span>
    </span>
    <span class="pbStatus"></span>
</div>

</body>
</html>
```

Nous partons du principe qu'une barre de progression est représentée par :

- 1 Un div jouant le rôle de conteneur pour la barre et son animation/image de droite (*spinner* ou icône de complétion).
- 2 Dans ce div, un premier span qui représente la barre à proprement parler, et un second qui fournira l'animation ou image.
- 3 Dans le span de la barre, un span gérant la barre colorée qui va progresser en largeur, et un autre fourniissant la représentation textuelle de la progression. En termes d'accessibilité, il est en effet bon d'avoir une représentation textuelle facilement repérable dans le flux du document.

Cette façon de faire n'est pas forcément la meilleure, mais elle permet de définir de multiples barres dans une même page, en fournissant simplement un id distinct pour chaque div, et en respectant juste les classes CSS indiquées à l'intérieur.

Listing 6-13 La feuille de styles pour ces barres de progression

```
div.progressBar {  
    font-family: sans-serif;  
    position: relative;  
    margin: 1em 0; width: 204px; height: 20px;  
}  
  
span.pbFrame {  
    position: absolute;  
    left: 0; top: 0; width: 180px; height: 16px;  
    border: 2px solid #444;  
    background-color: silver;  
}  
  
span.pbColorFill {  
    position: absolute;  
    left: 0; top: 0; height: 100%; width: 0;  
    z-index: 1;  
    background-color: green;  
}  
  
span.pbPercentage {  
    position: absolute;  
    left: 0; top: 0; height: 100%; width: 100%;  
    z-index: 2;  
    font-size: 1em;  
    line-height: 16px;  
    font-weight: bold;  
    text-align: center;  
}
```

```
span.pbPercentage.over50 {  
    color: white;  
}  
  
span.pbStatus {  
    position: absolute;  
    background-repeat: no-repeat;  
    top: 2px; width: 16px; height: 100%;  
    z-index: 2;  
}  
  
span.pbStatus.working {  
    background-image: url(spinner.gif);  
}  
  
span.pbStatus.done {  
    background-image: url(ok.png);  
}
```

Si certains aspects vous échappent, allez faire un tour à l'annexe B. Elle vous rappellera entre autres les concepts qui sous-tendent le positionnement et le modèle de boîtes, éclairant l'utilisation faite ici de `position` et `z-index`.

Notez l'emploi des classes multiples qui aide à bien séparer le comportement (changement d'état dans le temps) de l'aspect (images et couleurs différentes). En effet, dans le script ci-après, on ne touche pas directement au style, on se contente de manipuler les classes assignées aux éléments.

Enfin, il reste notre script client.

Listing 6-14 Le script client, qui dès le chargement interroge jusqu'à 10 fois par seconde

```
function getRequester() {  
    ...  
} // getRequester  
INTERVAL = 100; ①  
var gProgressTimer = 0;  
  
function checkProgress(id) {  
    var node = $(id);  
    var filler = node.down('.pbColorFill'); ②  
    var percent = node.down('.pbPercentage');  
    var status = node.down('.pbStatus');  
    var firstHit = 0 == gProgressTimer; ③  
    if (!firstHit) {  
        window.clearTimeout(gProgressTimer);  
        gProgressTimer = 0;
```

```

    } else {
        percent.removeClassName('over50');
        status.removeClassName('done').addClassName('working');
    }
var requester = getRequester();
requester.onreadystatechange = function() {
    if (4 == requester.readyState && 200 == requester.status) {
        var progress = parseInt(requester.responseText, 10); ④
        if (100 <= progress)
            progress = 100;
        filler.style.width = progress + '%'; ⑤
        percent.update(progress + '%');
        // Blanc sur vert, c'est plus joli... :-)
        if (progress > 50 && !percent.hasClassName('over50'))
            percent.addClassName('over50');
        if (100 == progress) { ⑥
            status.removeClassName('working').addClassName('done');
        } else
            gProgressTimer = window.setTimeout(
                ↳ 'checkProgress("' + id + '")', INTERVAL);
    }
};
requester.open('GET', '/whatsup', true);
requester.send(null);
} // checkProgress

document.observe('dom:load', checkProgress.curry('progress'));

```

- ① Intervalle entre requêtes de progression : 100 ms.
- ② Obtention des nœuds du DOM qui nous intéressent d'après leur classe, à l'intérieur du div identifié par id.
- ③ Distinction entre le premier appel (juste après le chargement de la page) et les suivants, histoire de pouvoir réutiliser la barre sans recharger la page, qui sait ?
- ④ On se protège contre « 08 », « 09 », etc. et... contre un serveur mal fichu !
- ⑤ En définissant une largeur en %, on s'adapte automatiquement à la taille du span conteneur.
- ⑥ Si on a fini, exit le *spinner*, voici le sceau de bonne fin ! Sinon, n'oublions pas de redemander une requête pour bientôt.

FICHIERS D'EXEMPLE Les images

Sur cet exemple, des fichiers vous manquent : les images spinner.gif et ok.png. Vous pouvez vous les procurer dans l'archive contenant tout le code de cet ouvrage, disponible en ligne sur le site des éditions Eyrolles.

Il ne vous reste plus qu'à lancer votre serveur (avez-vous pensé à arrêter le précédent ?), et à naviguer sur <http://localhost:8042/>. Et là, joie !

Figure 6–9

Quelques étapes de la progression, jusqu'à la fin



Voilà qui a tout de même une autre allure qu'un simple champ de formulaire !

Fragments de page prêts à l'emploi : réponse XHTML

Un mécanisme très fréquemment rencontré consiste à produire un fragment XHTML côté serveur, et le renvoyer au client. Ce dernier va « incruster » le fragment à l'endroit approprié.

Bien sûr, cela implique une légère intrusion de la couche présentation dans la couche métier, mais avec les bonnes méthodologies, ce n'est pas très grave. Après tout, la couche présentation est générée côté serveur pour des pages dynamiques. Il suffit de ne pas mettre de XHTML en dur dans le code pour rester plutôt propre, et justement, nous avons déjà vu ERb pour réaliser cela dans notre exemple !

Créez un nouveau répertoire, et comme d'habitude, copiez-y docroot, prototype.js et client.js, que vous réduirez à son getRequester. Nous allons aussi repartir de notre index.html précédent pour le squelette de la page principale.

Notre exemple sera un formulaire de commentaire, par exemple dans le cadre d'un blog. Le commentaire sera envoyé en Ajax, et cette fois-ci, nous ferons l'effort côté serveur permettant un envoi en POST, ce qui est tout de même plus conforme aux règles du W3C pour les actions censées modifier la couche serveur.

Le serveur générera le bloc XHTML de représentation du commentaire (pré-entièrement sauvé), et le renverra pour insertion. Nous utiliserons Prototype pour réaliser l'insertion en question.

Voici d'abord notre code serveur.

Listing 6-15 Notre script serveur, capable de traiter le POST comme le GET

```
#!/usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

template_text = File.read('commentaire.rhtml')
comment = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')
server.mount_proc('/add_comment') do |request, response|
  post = 'POST' == request.meta_vars['REQUEST_METHOD'] ①
  params = post ? CGI::parse(request.body) : request.query
  params.each { |k, v| params[k] = v[0] } if post
  # PASSEZ les paramètres, sous peine d'erreur de nil
  name = CGI::escapeHTML(params['name'])
  email = CGI::escapeHTML(params['email'])
  # Formatage simple : les textes séparés par des lignes vierges
  # sont des paragraphes et on supprime les paragraphes vides.
  text = CGI::escapeHTML(params['comment'])
  text.gsub!(/(\r?\n){2,}/, "</p>\n<p>")
  text = ('<p>' + text + '</p>').gsub(</p>\s*</p>/, '')
  response['Content-Type'] = 'text/html'
  response.body = comment.result(binding)
end

trap('INT') { server.shutdown }

server.start
```

Les 3 lignes de code en ① permettent de gérer tant le POST que le GET. Dans la mesure où vous n'utiliserez normalement jamais WEBrick directement en production (préférez, largement, Ruby on Rails !), on n'entrera pas dans les détails. Les habitués des scripts côté serveur de type CGI s'y retrouveront...

Le modèle de commentaire est très simple.

Listing 6-16 Le modèle de commentaire

```
<div class="comment">
    <h3><a href="mailto:<%= email %>"><%= name %></a>
    ↵ @ <%= Time.now.strftime('%H:%M') %></h3>
        <div class="commentText">
            <%= text %>
        </div>
</div>
```

Voici à présent notre page HTML cliente.

Listing 6-17 La page cliente, avec un formulaire plutôt sémantique

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
          charset=iso-8859-15" />
    <title>Commentaires</title>
    <link rel="stylesheet" type="text/css" href="client.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Commentaires</h1>

<p>Ajoutez un commentaire ci-dessous. En le soumettant, il sera envoyé en
arrière-plan au serveur, et sa représentation XHTML sera renvoyée puis
insérée dynamiquement dans cette page.</p>

<form id="commentForm" method="post" action="/add_comment">
    <fieldset>
        <legend>Un commentaire ?</legend>
        <p>
            <label for="edtName" accesskey="N">Nom</label>
            <input type="text" id="edtName" name="name" tabindex="1" />
        </p>
        <p>
            <label for="edtEmail" accesskey="C">Courriel</label>
            <input type="text" id="edtEmail" name="email" tabindex="2" />
        </p>
        <p class="comment">
            <label for="memComment" accesskey="0">Commentaire</label>
            <textarea id="memComment" name="comment" cols="40"
                      rows="5" tabindex="3">Votre commentaire ici</textarea>
        </p>
    </fieldset>
</form>
```

```
<p>
    <input type="submit" value="Envoyer !" tabindex="4" />
</p>
</fieldset>
</form>

<div id="comments"></div>

</body>
</html>
```

Notez que dans une application professionnelle, on fournirait des mécanismes permettant de traiter le formulaire normalement côté serveur, plutôt qu'en Ajax uniquement, à des fins d'accessibilité et de compatibilité maximales : c'est la raison pour laquelle notre formulaire est correctement paramétré.

Le div final servira de conteneur pour les insertions dynamiques des blocs XHTML des commentaires.

À présent, voici le script client.

Listing 6-18 Le script client, qui construit la requête POST et insère le résultat

```
function bindForm() {
    $('#commentForm').observe('submit', switchToAjax);
} // bindForm

function getRequester() {
...
} // getRequester

function switchToAjax(e) {
    e.stop();
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            $('#comments').insert(requester.responseText); ①
        }
    };
    requester.open('POST', '/add_comment', true);
    requester.send(this.serialize());
} // switchToAjax

document.observe('dom:loaded', bindForm);
```

L'appel Prototype en ① ajoute le fragment XHTML fourni dans la réponse dans l'élément comments, après son contenu existant. Par ailleurs, en POST, les paramètres sont par défaut encodés comme dans une URL, mais sont dans le corps de la requête. Utiliser serialize() sur le formulaire ② est décidément bien pratique...

Pour que ce soit joli, reste une petite CSS (il ne s'agit véritablement que d'esthétique, le script laissant les styles tranquilles).

Listing 6-19 La feuille de styles employée

```
form#commentForm fieldset {  
    background: #ccc;  
    width: 60ex;  
    border: 1px solid #444;  
}  
  
form#commentForm p {  
    position: relative;  
    height: 2.2em;  
}  
  
form#commentForm input#edtName, form#commentForm input#edtEmail,  
form#commentForm textarea#memComment {  
    font: inherit;  
    position: absolute;  
    left: 14ex; right: 0; top: 0; bottom: 0.5em; width: 44ex  
}  
  
form#commentForm p.comment {  
    height: 6.2em;  
}  
  
div.comment {  
    font-family: sans-serif;  
    width: 60ex; padding: 0 1ex; margin: 1em 0;  
    border: 1px solid #990;  
    background: #ffc;  
}  
  
div.comment h3 {  
    font-size: 100%;  
}  
  
div.comment h3 a {  
    color: maroon;  
}  
  
div.comment div.commentText {  
    border-left: 0.5ex solid gray;  
    margin-left: 1ex; padding-left: 1ex;  
    font-style: italic;  
}
```

Et voici un exemple après avoir saisi trois séries de valeurs et pressé le bouton d'envoi à chaque fois ; la page ne s'est jamais rechargée !

Figure 6-10
La page après trois
saisies envoyées

The screenshot shows a Mozilla Firefox browser window with the title "Commentaires - Mozilla Firefox". The address bar displays "http://localhost:8042/". The main content area contains a form titled "Commentaires". The form includes fields for "Nom" (Christophe), "Courriel" (tdd@example.com), and "Commentaire" (containing the text "Exemple d'encodage du HTML : <hollo>< > & ' " Rien ne passe !"). Below the form is a button labeled "Envoyer !". Underneath the form, there are three previous comments from "Christophe" at 13:50, 13:50, and 13:51, each with a different test message. At the bottom of the page, a status bar says "Terminé".

Pour finir avec XHTML, sachez que la notion globale de communication asynchrone HTTP renvoyant un fragment de page est si répandue qu'un microformat lui est dédié : AHAH. Les microformats visent à fournir une ou plusieurs fonctionnalités en se basant sur des formats et standards existants, utilisés de façon innovante et formalisée. Vous en saurez plus en consultant le site dédié aux microformats : <http://www.microformats.org/about/>, et la page spécifique au microformat AHAH : <http://www.microformats.org/wiki/rest/ahah>.

Dans la cour des grands : XPath pour traiter des données XML complexes

Tout cela est bel et bon mais on peut se demander pourquoi on ne renvoie pas de XML. Dans Ajax, le X représente XML, et renvoyer une grappe XML peut impressionner. Après tout, tout est à base de XML aujourd'hui : enlevez XML et J2EE n'existe plus, tout comme SOAP et OpenDocument.

Certes. Mais le plus grand danger de XML résulte justement de son immense popularité : on veut en mettre partout. C'est un peu comme Ajax aujourd'hui. Et bien que la pulsion du « tout XML » soit en baisse (heureusement, depuis 1998, les passions se sont calmées), on en trouvera toujours pour miser dessus sans rime ni raison. Le XML n'est pourtant pas incontournable : divers frameworks fonctionnent très bien sans lui.

Ceci dit, je n'écrirais pas une section ainsi nommée si je voulais envoyer XML aux orties ; il existe certains cas de figure où XML est approprié. Il s'agit principalement de cas où les données à récupérer sont particulièrement riches, ou présentent certaines caractéristiques de complexité (par exemple, un nombre inconnu d'éléments, ou des éléments aux noms inconnus à l'avance). Même s'il reste possible de traiter de tels cas autrement, avec JSON par exemple, XML va briller si on doit « fouiller » dans les données de façon un tant soit peu puissante.

En effet, dès lors qu'on dispose d'une grappe XML, on peut utiliser des technologies comme XPath ou encore XSLT pour réaliser des recherches complexes ou des transformations puissantes sur cette grappe. Pour de tels traitements, JSON ne peut tout simplement pas concurrencer XML.

Cette perspective est tout à fait alléchante, mais il convient de mettre un bémol, comme souvent dans l'univers des technologies web côté client : celui de la compatibilité des navigateurs.

Le W3C a défini les spécifications XSLT et XPath ainsi que leurs cousines liées au DOM (par exemple DOM niveau 3 XPath), qui définissent des interfaces telles que `XSLTProcessor`. Ces interfaces sont censées, à terme, être proposées par des objets accessibles en JavaScript, par exemple `document`. Cependant, la prise en charge de ces standards varie grandement d'un navigateur à l'autre.

Dans les exemples qui vont suivre, nous allons nous concentrer sur XPath, pour voir comment extraire quelques informations d'une grappe XML sans la parcourir manuellement *via* les possibilités classiques du DOM (mais si, vous savez bien : `firstChild`, `nextSibling`, `nodeType`, `nodeValue`, et les autres...). Il nous faut donc un navigateur prenant en charge XPath (presque tous, en interne), mais aussi et surtout le DOM niveau 3 XPath.

C'est là que les choses se gâtent : si Firefox (et donc Camino) et Opera (à partir de la version 9) répondent présents, on tombe actuellement à plat sur MSIE, Safari et Konqueror. Il est certes permis d'espérer du changement côté Safari et Konqueror (qui se talonnent toujours de près, n'étant pas sans rapport), mais on sait que côté MSIE, il faudra attendre au grand minimum la version 8, puisque la 7, encore une fois, ne prévoit aucune amélioration significative sur JavaScript ou le DOM.

Utiliser une technologie indisponible sur MSIE peut en refroidir plus d'un, en particulier pour un service Extranet ou Internet. Mais rassurez-vous : des palliatifs existent, sous forme de bibliothèques JavaScript simulant la fonctionnalité. Nous verrons donc d'abord un exemple avec du code « natif » (prise en charge du DOM niveau 3 XPath), qu'il vous faudra nécessairement tester sur Firefox, Camino ou Opera 9. Par la suite, nous adapterons sa couche client pour réaliser l'équivalent au moyen d'une bibliothèque tierce partie, en l'occurrence GoogleAJAXSLT (voilà un nom qui noue la langue).

Vite et bien : utilisation de DOM niveau 3 XPath

Nous allons implémenter une petite fonction toute bête : une liste de flux Atom pour quelques blogs reconnus, dont nous irons chercher dynamiquement le nombre d'articles. Théoriquement, cela pourrait se passer intégralement côté client, mais cela implique une configuration de sécurité particulière, qui varie sensiblement d'un navigateur à l'autre.

Pour laisser ces problèmes de côté pour l'instant (nous aurons tout le temps de les examiner aux chapitres 9 et 10), nous allons implémenter la récupération des contenus en passant par notre couche serveur, qui jouera le rôle de l'intermédiaire.

Créez un nouveau répertoire et son sous-répertoire docroot, copiez-y `prototype.js` et `client.js`, dont on récupérera le `getRequester`, ainsi que le `index.html` pour son squelette.

Voici notre couche serveur, qui a en outre l'avantage de forcer un type de réponse `text/xml`, afin de garantir la construction d'un DOM côté client dans la propriété `responseXML` du requêteur :

Listing 6-20 Notre serveur, simple « proxy » de chargement des flux

```
#!/usr/bin/env ruby

require 'net/http'
require 'uri'
require 'webrick'
include WEBrick
```

```

FEEDS = {
  'Standblog' => 'http://standblog.org/blog/feed/atom',
  'Formats Ouverts' => 'http://formats-ouverts.org/atom.php',
  'IEBlog' => 'http://blogs.msdn.com/ie/atom.xml'
}

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/get_feed') do |request, response|
  # Critique pour pouvoir utiliser requester.responseXML !
  response['Content-Type'] = 'text/xml'
  feed = FEEDS[request.query['feed']]
  response.body = Net::HTTP.get(URI.parse(feed))
end

trap('INT') { server.shutdown }

server.start

```

Attention ! Si vous êtes derrière un proxy, il vous faut ajuster ce code, en remplaçant :

Net::HTTP.get

par :

Net::HTTP.Proxy('votre_hote_proxy', votre_port_proxy).get

Rien de bien compliqué... Si le URI.parse vous intrigue, sachez simplement qu'une URL est un type particulier d'URI.

Voyons à présent notre page HTML, très simple (pour simplifier le code de cet exemple, on n'a pas rendu la liste des flux dynamique).

Listing 6-21 La page cliente, avec le formulaire de choix de flux

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR" xml:lang="fr-
FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-15" />
  <title>Dernières nouvelles (XML/XPath)</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>

```

```
<body>

<h1>Dernières nouvelles (XML/XPath)</h1>

<form id="feedForm">
    <p>
        <label for="cbxFeed" accesskey="F">Flux Atom</label>
        <select id="cbxFeed" name="feed">
            <option>Standblog</option>
            <option>Formats Ouverts</option>
            <option>IEBlog</option>
        </select>
        <input type="button" id="btnProcessFeed"
               value="Articles récents ?"
               accesskey="A" />
    </p>
    <p id="status">&nbsp;</p>
</form>

</body>
</html>
```

Il nous faut aussi un brin de CSS pour afficher le traitement en cours et son résultat.

Listing 6-22 La petite feuille de styles

```
p#status {
    height: 16px;
    font-family: sans-serif;
    color: gray;
    background-repeat: no-repeat;
}

p#status.working {
    background-image: url(spinner.gif);
}
```

Un flux Atom est un document XML avec un élément racine `feed` contenant, entre autres, un élément `entry` par article. Il ne s'agit pas forcément de tous les articles du blog, juste des derniers : la taille peut être configurée dans le logiciel de blog.

Voici notre code JavaScript.

Listing 6-23 Le code client JavaScript, utilisant XPath

```
function bindButton() {
    $('#btnProcessFeed').observe('click', processFeed);
} // bindButton
```

```
function getRequester() {
    ...
} // getRequester

function processFeed() {
    var feed = $('#cbxFeed').serialize();
    var form = $('#feedForm');
    var status = $('#status');
    form.disable();
    status.update('').addClassName('working');
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            var data = requester.responseXML;
            var articleCount = data.evaluate(
                'count(//*[name()="entry"])', data, null,
                XPathResult.NUMBER_TYPE, null).numberValue;
            status.removeClassName('working');
            form.enable();
            status.update(articleCount + ' article(s).');
        }
    };
    requester.open('GET', '/get_feed?' + feed, true);
    requester.send(null);
} // processFeed

document.observe('dom:loaded', bindButton);
```

Remarquez les petites précautions que nous prenons pour empêcher le déclenchement parallèle de plusieurs requêtes : nous désactivons le formulaire le temps du traitement et affichons bien sûr le *spinner* pour patienter.

Par ailleurs, il convient de faire la lumière sur l'invocation quelque peu obscure de la méthode `evaluate` :

- Il s'agit d'une méthode DOM niveau 3 XPath, implémentée par tout élément qui implémente également l'interface DOM Document.
- Elle prend 5 paramètres obligatoires :
 1. L'expression XPath.
 2. Le noeud servant de contexte à cette expression (souvent le document sur lequel on appelle `evaluate`, en particulier pour une expression démarrant par `/`).
 3. Un solveur éventuel d'espaces de noms (inutile ici, donc `null`).
 4. Le type de résultat souhaité. Parmi les nombreux types, nous choisissons `NUMBER_TYPE`, déclaré dans la classe `XPathResult`, ce qui convient au résultat d'une fonction `count`.

5. Un objet résultat existant qui sera alors recyclé. Nous n'en avons pas sous la main (ce serait différent si nous bouclions autour d'`evaluate`, par exemple), aussi nous fournissons `null`.

- Elle renvoie un objet `XPathResult` configuré d'après le 4^e argument. Dans le cas d'un résultat numérique, on exploite la propriété `nodeValue`.

Enfin, l'expression employée dans la fonction `count` signifie : « à tout niveau de profondeur du document (`//`), les éléments dont le nom est `entry` ». En XPath, on peut exprimer cela bien plus simplement, mais Firefox et Safari semblent avoir un souci, très surprenant d'ailleurs, avec « `//entry` »...

Et voilà ! Une analyse qui nous aurait pris quelques lignes de JavaScript (beaucoup même, si nous n'avions pas Prototype) est ici très courte. Et encore, il ne s'agit que d'une analyse simple. On pourrait imaginer ne sortir que les articles dont le titre comporte un certain texte et mis à jour dans la semaine écoulée, en augmentant à peine l'expression XPath...

En simulant : utilisation de GoogleAJAXSLT

On l'a vu, cette fonctionnalité n'est pour l'instant disponible, en natif, que sur Firefox, Camino et Opera 9 (et ultérieurs, évidemment). Ce qui laisse tout de même de côté Safari, principal navigateur pour les utilisateurs de Mac OS X, Konqueror, très prisé des aficionados de KDE, et surtout MSIE, qui couvre tout de même encore plus de 70 % du marché, monopole oblige.

Pour disposer des mêmes fonctionnalités sur ces navigateurs, il faut pour l'instant avoir recours à des bibliothèques JavaScript tierces. Elles sont légion, mais l'une des plus prometteuses (et dont les performances sont raisonnables, même sur les quelque 21 Ko du flux Atom de l'IEBlog !) est GoogleAJAXSLT (prononcez « Google Ajax S-L-T », ou à l'anglaise : « ... S-L-Ti »).

Cette bibliothèque, écrite par le *googlien* Steffen Meschkat, fournit un équivalent pas tout à fait complet, mais largement suffisant, des spécifications XPath et XSLT, directement en JavaScript. Aucune dépendance externe n'est requise (par exemple, la possibilité de recourir aux ActiveX sous MSIE, ou même simplement la présence de la bibliothèque Prototype), si ce n'est la prise en charge par le navigateur de l'essentiel de DOM niveau 2, ce qui nous assure un bon fonctionnement sur la quasi-totalité des navigateurs récents.

La bibliothèque se trouve à l'adresse : <http://code.google.com/p/ajaxslt/>, et est aussi présente dans l'archive des codes source de ce livre, dans une version très légèrement retouchée pour corriger un petit défaut, dans `xpath.js` de la fonction `translate`, utilisée au chapitre 9. C'est cette dernière version que nous utiliserons. Déposez les fichiers de script mentionnés ci-dessous dans un sous-répertoire `ajaxslt` de votre docroot.

Son utilisation diffère finalement assez peu du code que nous avons écrit pour l'exemple natif. Bien entendu, le serveur ne change pas, tout comme la page cliente (aux scripts près, comme indiqué dans un instant) et la feuille de styles. Il faut en revanche charger des scripts supplémentaires, fournis par la bibliothèque. Il est préférable de les charger dans l'ordre suivant : util, xmltoken, dom et xpath.

L'en-tête de notre `index.html` prend donc la forme suivante.

Listing 6-24 Nouvel en-tête pour notre page cliente

```
<head>
    <meta http-equiv="Content-Type" content="text/html;
        ↵ charset=iso-8859-15" />
    <title>Dernières nouvelles (XML/XPath avec GoogleAJAXSLT)</title>
    <link rel="stylesheet" type="text/css" href="client.css" />
    <script type="text/javascript" src="ajaxslt/util.js"></script>
    <script type="text/javascript" src="ajaxslt/xmltoken.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt>xpath.js"></script>
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
```

Ceci fait, il nous reste à ajuster le code client. Les modifications sont chirurgicales :

- 1 Désactivez la journalisation du moteur (affichage dans un pavé à position fixe des grandes étapes du traitement XPath), à l'aide d'une variable globale.
- 2 Remplacez l'appel à `document.evaluate` par la construction d'un contexte pour le XML et un appel à `xpathParse`.
- 3 Remplacez la propriété `nodeValue` par la méthode `nodeValue()`.

C'est tout ! Voici l'extrait modifié du code client, avec les altérations mises en évidence (notez que la variable se termine par un double souligné).

Listing 6-25 La portion modifiée de client.js

```
if (4 == requester.readyState && 200 == requester.status) {
    var data = requester.responseXML;
    logging__ = false;
    var ctx = new ExprContext(data);
    var articleCount = xpathParse(
        ↵ 'count(//entry)').evaluate(ctx).nodeValue();
    status.removeClassName('working');
    form.enable();
    status.update(articleCount + ' article(s).');
}
```

Et voilà, tout se passe comme avant, mais cette fois-ci cela fonctionne aussi sur MSIE, Safari, Konqueror...

Piloter la page en renvoyant du JavaScript

Générer du JavaScript côté serveur ! Aurais-je perdu l'esprit ? Je vous imagine aisément froncer les sourcils d'un air réprobateur devant une telle méprise apparente des règles du jeu.

Et pourtant, dans une application complexe, les comportements à adopter dans la page client au retour d'une requête Ajax peuvent être non seulement complexes, mais aussi varier d'une requête à l'autre. Dans tel cas, il faudra ajouter un fragment XHTML, dans tel autre lancer un requêteur Ajax périodique, masquer le message de notification et modifier un texte, etc.

Le fait est que traiter tous ces cas de figure dans le JavaScript de la fonction de rappel (ou dans des fonctions séparées que cette dernière appellera) conduit à un code JavaScript volumineux côté client. La factorisation est potentiellement difficile, dans la mesure où on souhaite avoir des fichiers JS séparés de nos pages afin de favoriser le travail des caches. Bref, c'est vite une galère !

En générant le JavaScript côté serveur (quelle que soit la technologie employée), on a beaucoup plus de flexibilité : on peut renvoyer le bout de script exactement adapté à la situation. Ce script peut être autonome (à Prototype près, le plus souvent), ou reposer sur des variables et fonctions dont il connaît la disponibilité côté client.

La génération du morceau de script peut varier en difficulté : ainsi, dans l'exemple ci-après, nous allons le générer à la main, ce qui n'est ni très élégant ni très facile. Dans certains frameworks, cette fonction est déjà là. Ruby on Rails, notamment, a introduit en 2006, dans sa version 1.2, les *templates RJS*, qui permettent d'écrire en quelques lignes de Ruby un modèle de réponse Ajax qui sera traduit à la volée en JavaScript portable basé sur Prototype ! Bonheur !

Pour cet exemple, nous allons reprendre notre barre de progression et l'adapter pour faire en sorte que le code côté client n'ait plus de logique algorithmique pour le traitement du pourcentage : c'est le serveur qui déterminera le code à exécuter suite à l'envoi de la requête Ajax. Le côté client se contentera de mettre à disposition les objets nécessaires (composants de la barre de progression concernée), et d'évaluer le code JavaScript renvoyé par le serveur.

Recopiez complètement votre répertoire de travail pour l'exemple de la barre de progression. Nous n'allons en changer que deux fichiers : le serveur et le script.

Commençons par le script : il s'agit simplement de remplacer, dans le traitement du retour de requête Ajax, l'ancien code suivant :

```
var progress = parseInt(requester.responseText, 10);
if (100 <= progress)
    progress = 100;
filler.style.width = progress + '%';
percent.update(progress + '%');
// Blanc sur vert, c'est plus joli... :-)
if (progress > 50 && !percent.hasClass('over50'))
    percent.addClass('over50');
if (100 == progress) {
    status.removeClassName('working').addClassName('done');
} else
    gProgressTimer = window.setTimeout(
        => 'checkProgress("' + id + '")', INTERVAL);
```

par celui-ci :

```
eval(requester.responseText);
```

C'est tout ! La délégation, ça a du bon, vous ne trouvez pas ?

Passons maintenant côté serveur, où le code est forcément un peu plus lourd. Au lieu de simplement renvoyer la valeur de progression, nous allons composer le JavaScript à exécuter sur le client. Voici l'intégralité du fichier.

Listing 6-26 Le serveur compose le fragment de JavaScript à exécuter

```
#!/usr/bin/env ruby

require 'webrick'
include WEBrick

progress = 0
colorChanged = false

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/whatsup') do |request, response|
    response['Content-Type'] = 'text/plain'
    progress += rand(5) + 1
    progress = 100 if progress > 100
    script = %{
```

```
filler.style.width = '#{progress}%';
percent.update('#{progress}%');
}
if progress > 50 and !colorChanged
  colorChanged = true
  script += %{
    percent.addClassName('over50');
  }
end
# Arrivé au bout ? Remise à zéro pour la prochaine fois ;)
if 100 == progress
  script += %{
    status.removeClassName('working').addClassName('done');
  }
  progress = 0
  colorChanged = false
else
  script += %{
    gProgressTimer = window.setTimeout(
      ↳ 'checkProgress"' + id + '"', INTERVAL);
  }
end
# Nettoyer le script (pensons aux yeux des débogueurs !)
# --> on vire l'indentation inutile et les lignes vides
script.gsub!(/\s+/, '')
response.body = script
end

trap('INT') { server.shutdown }

strand
server.start
```

La syntaxe `%{...}` encadre des littéraux textuels où la syntaxe `#{...}` permet d'incorporer des portions variables. Cela est fort utile pour manipuler des portions de code, ici JavaScript.

On remarque que notre script repose sur la disponibilité de variables `filler`, `percent` et `status`, qui sont effectivement définies par le script côté client avant d'appeler `eval` sur notre fragment.

Et voilà ! On peut lancer le serveur, afficher `http://localhost:8042/`, et voir la barre fonctionner comme avant ! Sauf que le JavaScript d'évolution pour chaque étape est composé sur le serveur.

Voici trois exemples de JavaScript retourné : avant 50 %...

```
filler.style.width = '16%';
percent.update('16%');
gProgressTimer = window.setTimeout('checkProgress("' + id + '")',
INTERVAL);
```

Au premier passage après 50 % :

```
filler.style.width = '51%';
percent.update('51%');
percent.addClassNames('over50');
gProgressTimer = window.setTimeout('checkProgress("' + id + '")',
INTERVAL);
```

En arrivant à 100 % :

```
filler.style.width = '100%';
percent.update('100%');
status.removeClassNames('working').addClassNames('done');
```

Vous voyez : rien d'autre que le strict nécessaire.

JSON : l'idéal pour des données structurées spécifiques

JSON signifie *JavaScript Object Notation*. Il s'agit d'une représentation formelle d'objets de complexité quelconque, en utilisant uniquement des syntaxes disponibles en JavaScript, principalement pour décrire des tableaux et des objets. Le format est documenté sur <http://www.json.org>, qui fournit également des liens vers les principales bibliothèques JSON pour de nombreux langages.

Les bases de JSON sont très simples :

- Un objet est décrit entre accolades ({}), sous forme d'une série de paires nom + valeur, séparées par des virgules. Le nom est fourni en tant que chaîne de caractères (donc encadré par ' ou "), et la valeur peut être n'importe quel littéral JavaScript (ou presque) : chaîne de caractères, nombre, booléen, tableau, null, etc.
- Un tableau est décrit entre crochets ([]), ses éléments sont séparés par des virgules.

Par ailleurs, pour faire dans l'impeccable, nous allons également renvoyer un en-tête de type de contenu spécialisé, `application/json`, qui indique que notre retour est une donnée JSON.

Nous allons utiliser comme exemple un affichage de statistiques. Histoire de mélanger un peu les genres, nous allons afficher un pourcentage sans signification particulière

pour trois marchés internationaux : le CAC40, le NYSE et le NASDAQ, accompagné d'un commentaire tiré au hasard parmi certaines possibilités.

Nous avons ici plus qu'une simple donnée textuelle ; nous avons plusieurs occurrences d'une même information structurée, laquelle comprend trois parties :

- 1 le symbole du marché (cac40, nyse ou nasdaq) ;
- 2 le pourcentage pour ce marché ;
- 3 le texte de commentaire.

Nous allons donc devoir générer une représentation JSON ressemblant à ceci (l'indentation est juste là pour la lisibilité, elle n'a aucun impact en JavaScript) :

```
[  
  { 'symbol': 'cac40', 'percent': 78, 'comment': 'nervieux' },  
  { 'symbol': 'nyse', 'percent': 16, 'comment': 'calme' },  
  { 'symbol': 'nasdaq', 'percent': 43, 'comment': 'actif' }  
]
```

On peut repartir d'un exemple précédent avec barres de progression, car nous allons légèrement ajuster le code HTML et CSS pour correspondre à nos besoins.

Voici d'abord notre serveur.

Listing 6-27 Le serveur envoyant des données JSON

```
#!/usr/bin/env ruby  
  
require 'webrick'  
include WEBrick  
  
STOCKS = [ 'cac40', 'nyse', 'nasdaq' ]  
COMMENTS = [ 'actif', 'nervieux', 'calme', 'en folie !' ]  
  
server = HTTPServer.new(:Port => 8042)  
server.mount('/', HTTPServlet::FileHandler, './docroot')  
  
server.mount_proc('/stats') do |request, response|  
  response['Content-Type'] = 'application/json'  
  data = '['  
  STOCKS.each { |symbol|  
    comment = COMMENTS[rand(COMMENTS.size)]  
    data += "\n\t{ 'symbol': '#{symbol}', 'percent': #{rand(101)},  
      'comment': '#{comment}' },"  
  }  
  data.chop!  
  data += "\n]"  
  response.body = data  
end
```

```
trap('INT') { server.shutdown }

strand
server.start
```

La méthode `chop!` retire le dernier caractère d'une chaîne de caractères, ce qui nous permet d'éviter une virgule finale superflue qui, d'ailleurs, gênerait MSIE.

Côté client, les barres n'ont plus d'icône ou d'animation à leur droite. Chaque barre a une taille fixe (le remplissage varie évidemment en fonction du pourcentage), et le texte de commentaire prend le reste de la place. On a donc le code HTML suivant.

Listing 6-28 La page HTML client

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  > xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    > charset=iso-8859-15" />
  <title>Statistiques avec JSON</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>
  <h1>Statistiques avec JSON</h1>
  <p>Statistiques mises à jour toutes les 2 secondes.</p>
  <div class="progressBar" id="progress-cac40">
    <span class="pbFrame">
      <span class="pbColorFill"></span>
      <span class="pbPercentage">0%</span>
    </span>
    <span class="pbComment">n/a</span>
  </div>
  <div class="progressBar" id="progress-nyse">
    <span class="pbFrame">
      <span class="pbColorFill"></span>
      <span class="pbPercentage">0%</span>
    </span>
    <span class="pbComment">n/a</span>
  </div>
  <div class="progressBar" id="progress-nasdaq">
    <span class="pbFrame">
      <span class="pbColorFill"></span>
      <span class="pbPercentage">0%</span>
    </span>
```

```
<span class="pbComment">n/a</span>
</div>
</body>
</html>
```

Et la CSS ajustée que voici.

Listing 6-29 La nouvelle CSS

```
div.progressBar {
    font-family: sans-serif;
    position: relative;
    margin: 1em 0; height: 20px;
}

span.pbFrame {
    position: absolute;
    left: 0; top: 0; width: 20ex; height: 16px;
    border: 2px solid #444;
    background-color: silver;
}

span.pbColorFill {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 0%;
    z-index: 1;
    background-color: green;
}

span.pbPercentage {
    position: absolute;
    left: 0; top: 0; height: 100%; width: 100%;
    z-index: 2;
    font-size: 1em;
    line-height: 16px;
    font-weight: bold;
    text-align: center;
}

span.pbPercentage.over50 {
    color: white;
}
span.pbComment {
    position: absolute;
    left: 21ex; right: 0; top: 0; line-height: 1.5em;
    color: #444;
    z-index: 2;
}
```

Côté code client, il nous reste à demander toutes les 2 secondes, en mode asynchrone, des nouvelles de nos marchés au serveur. Et bien sûr, à traiter le résultat. On va voir que, JavaScript interprétant le code JSON comme des objets, cela donne un traitement plutôt pratique à écrire.

Listing 6-30 Le script d'invocation et de traitement des statistiques

```
INTERVAL = 2000;
var gLookupTimer = 0;

function getRequester() {
...
} // getRequester

function loadStats() {
    window.clearTimeout(gLookupTimer);
    var requester = getRequester();
    requester.onreadystatechange = function() {
        if (4 == requester.readyState && 200 == requester.status) {
            var data = $A(eval(requester.responseText)); ①
            updateStats(data);
            gLookupTimer = window.setTimeout('loadStats()', INTERVAL);
        }
    };
    requester.open('GET', '/stats', true);
    requester.send(null);
} // loadStats

function updateStats(data) {
    data.each(function(stock) { ②
        var bar = $('progress-' + stock.symbol);
        var filler = node.down('.pbColorFill');
        var percent = node.down('.pbPercentage');
        var comment = node.down('.pbComment');
        filler.style.width = stock.percent + '%';
        percent.update(stock.percent + '%');
        comment.update(stock.symbol.toUpperCase()
                       + ' ' + stock.comment);
        if (stock.percent > 50)
            percent.addClassNames('over50');
        else
            percent.removeClassNames('over50');
    });
} // updateStats

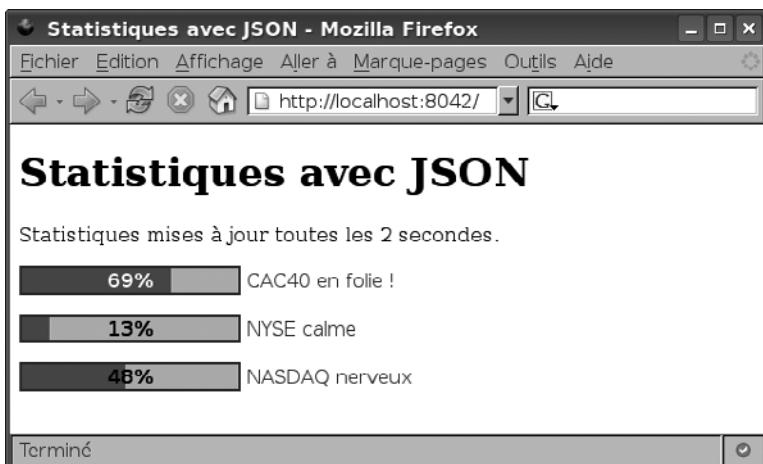
document.observe('dom:loaded', loadStats);
```

➊ On convertit le résultat en tableau « boosté » par Prototype et on passe la main à une fonction dédiée de traitement.

➋ stock est tour à tour chaque objet statistique. Remarquez comme on a accès directement à ses propriétés symbol, percent et comment !

Au final, on obtient par exemple ceci.

Figure 6-11
L'écran de statistiques



Nous en avons fini avec l'exploration des principaux types de réponse. J'aime à croire que ces exemples auront excité votre imagination et peut-être suggéré des débuts de réponse pour vos besoins réels.

Mais avant de vous jeter dans le code, souvenez-vous que tout ce que nous avons accompli jusqu'ici, nous l'avons réalisé sans tirer pleinement parti de Prototype et sans aucun framework supplémentaire. Attendez donc d'avoir lu les chapitres suivants, qui vont simplifier tout cela et ouvrir encore davantage de perspectives, avant de partir appliquer ces connaissances à vos développements.

7

Ajax tout en souplesse avec Prototype

Nous avons jusqu'ici utilisé Ajax « à la main » : nous avons manipulé `XMLHttpRequest` directement, ce qui engendre une certaine complexité, ou tout au moins un volume de code important, en particulier si on considère la simplicité du service qu'il rend. Dans ce chapitre, nous allons sauter le pas et tirer parti des utilisations prédéfinies d'Ajax proposées par Prototype.

Avec la version 1.6, la principale nouveauté réside dans l'utilisation de la classe `Ajax.Response` pour représenter les réponses HTTP reçues *via* Ajax, afin notamment de fournir une meilleure prise en charge de JSON.

Prototype encore à la rescousse

En réalisant les exemples précédents, on s'aperçoit qu'on écrit pas mal de code de « bas niveau » : l'algorithme sinueux d'obtention d'un requêteur, les multiples appels pour paramétrier et envoyer la requête, et ceci de différentes façons suivant qu'on est en POST ou en GET...

Comme au chapitre 4, Prototype vient à notre rescousse, cette fois-ci pour nous permettre d'écrire facilement du code reposant sur Ajax, sans nous soucier des incompatibilités éventuelles entre navigateurs, et d'une façon plus orientée objet, plus élégante.

Ajax.Request, c'est tellement plus simple !

Prototype fournit une aide considérable à l'utilisation d'Ajax au travers de trois classes principales et de quelques possibilités annexes, un peu plus ésotériques. Nous allons commencer par le comportement de base : l'exécution d'une unique requête et le traitement de sa réponse.

Commençons par une utilisation appliquée, en ajustant notre exemple de sauvegarde automatique, issu du chapitre précédent. Nous attendrons la prochaine section pour dresser un inventaire détaillé des possibilités.

Recopiez votre arborescence de travail pour l'exemple de sauvegarde automatique que nous avons réalisé comme première application d'un type de réponse « texte simple ». Nous allons simplement modifier le script dédié, dans le fichier `client.js`. Ces modifications consistent à :

- 1 Retirer notre fonction `getRequester` (enfin !).
- 2 Modifier le code de `syncName` pour tirer parti de Prototype au lieu de récupérer, configurer, exécuter et traiter manuellement notre requête et sa réponse.

Voici le texte intégral de notre nouveau `client.js`.

Listing 7-1 Le nouveau script client pour notre sauvegarde automatique

```
function bindTextField(e) {
    $('#edtName').observe('blur', syncName);
} // bindTextField
```

```
function syncName(e) {
    var req = new Ajax.Request('/save_name',
    {
        method: 'get',
        parameters: $('edtName').serialize(),
        onComplete: function(response) {
            if (!(req.success() && /\^200/.test(response.responseText)))
                alert('Le nouveau nom n\'a pas pu être sauvegardé !');
        }
    });
} // syncName

document.observe('dom:loaded', bindTextField);
```

Comme vous le voyez, c'est assez court. Les lecteurs attentifs (donc vous) auront remarqué l'absence d'un appel de type `send()`. La raison en est simple : un objet `Ajax.Request` déclenche sa requête immédiatement après initialisation.

Celle-ci comprend deux arguments : l'URL de base à invoquer (sans les paramètres, contrairement à notre code manuel précédent) et un tableau associatif d'options. Celles-ci sont nombreuses, mais contentons-nous pour l'instant d'expliciter les trois que notre exemple utilise :

- `method` permet d'indiquer la méthode HTTP à utiliser pour la requête. Par défaut à `post`, elle est ici modifiée en `get`, conformément aux exigences de notre couche serveur, qui n'a pas changé.
- `parameters` permet de préciser des paramètres pour une requête de type `get`, lesquels seront placés dans la *query string* de l'URL (le point d'interrogation est ajouté automatiquement). Il peut s'agir d'une chaîne déjà URL-encodée, d'un tableau associatif simple ou d'un objet Hash.
- `onComplete` fournit une fonction de rappel qui sera invoquée une fois la réponse terminée (quel que soit l'état du requêteur).

Les fonctions de rappel ne sont pas automatiquement *bindées* sur le requêteur. Il nous appartient donc soit de conserver une référence, comme nous le faisons dans le code précédent avec la variable `req`, soit d'utiliser la propriété `request` de l'objet `response` passé en argument.

À part ça, rien ne change ! Sur un exemple aussi simple, on ne se rend pas bien compte de la puissance offerte par `Ajax.Request`. Aussi, la section suivante explore cette classe plus en détail, afin de vous mettre l'eau à la bouche...

Plus de détails sur Ajax.Request

Tous les requêteurs Ajax (donc `Ajax.Request` et `Ajax.Updater`) partagent les méthodes suivantes

Tableau 7–1 Les méthodes communes à tous les requêteurs Ajax

Méthode	Description
<code>getHeader(nom)</code>	Renvoie un en-tête de réponse (n'a donc de sens qu'une fois que les en-têtes ont été reçus), en utilisant <code>null</code> si l'en-tête n'existe pas ou si une erreur survient dans sa récupération.
<code>isSameOrigin()</code>	Détermine si la requête Ajax observe la <i>Same Origin Policy</i> ou non. Par défaut, les navigateurs (hormis MSIE) utilisent cette politique et n'autoriseront donc pas une requête inter-domaines, mais des standards sont en cours de mise au point pour permettre cela de façon sécurisée, et MSIE le permet de toutes façons d'entrée de jeu. Cette méthode est notamment utilisée en interne pour la nouvelle prise en charge des réponses JSON, comme nous le verrons plus loin.
<code>success</code>	Permet de définir les options de requête, en définissant d'abord les valeurs par défaut décrites plus bas, puis en appliquant celles fournies sous forme d'un tableau associatif ou d'un objet. Détermine si la réponse est classée comme un succès (code de réponse HTTP dans la famille 2xy) ou non.

Tous les requêteurs Ajax partagent par ailleurs un certain nombre d'options. En voici la liste.

Tableau 7–2 Les options disponibles pour les requêteurs Ajax

Option	Défaut	Description
<code>asynchronous</code>	<code>true</code>	Mode de synchronisation de la requête. On n'y touche généralement pas.
<code>method</code>	<code>'post'</code>	Méthode HTTP à utiliser. Valeurs possibles garanties : <code>'get'</code> et <code>'post'</code> . Les standards en cours d'élaboration exigent néanmoins aussi le support de <code>'put'</code> , <code>'delete'</code> et <code>'head'</code> , notamment pour faciliter l'accès Ajax à des API de type REST (pour le moment, dans de tels cas, Prototype fera exceptionnellement preuve d'affinité avec une couche serveur spécifique, en l'occurrence Ruby on Rails, en injectant un paramètre synthétique <code>_method</code> avec la valeur fournie, et en basculant sur la méthode <code>post</code> afin de rester portable). La casse est sans importance, elle sera repassée en minuscules de toutes façons.
<code>contentType</code>	(voir texte)	Le type d'encodage pour les paramètres envoyés. Dans la mesure où l'on ne peut pas envoyer de fichiers directement en Ajax à ce jour, le type par défaut, qui est aussi celui de l'attribut <code>enctype</code> de <code><form></code> , est généralement conservé : <code>'application/x-www-form-urlencoded'</code> .

Tableau 7–2 Les options disponibles pour les requêteurs Ajax (suite)

Option	Défaut	Description
encoding	'UTF-8'	Là aussi, il vaut mieux travailler en UTF-8 de bout en bout quand on fait des applications web. Mais si vous êtes coincés dans une architecture ISO-8859-15 (ou pire encore : ISO-8859-1 et son absence de caractères comme œ ou €...), ajustez ce paramètre en fonction.
evalJS	true	Détermine si les réponses de type MIME JavaScript doivent être évaluées automatiquement ou non ; nous y reviendrons plus loin.
evalJSON	true	Détermine si les réponses de type MIME JSON doivent être évaluées automatiquement ou non ; là encore, nous y reviendrons.
onCreate	undefined	Fonction de rappel à la création du requêteur. N'a d'intérêt que pour les rappels globaux, que nous verrons en fin de chapitre. Par souci de cohérence avec les autres rappels, <code>onCreate</code> est néanmoins aussi disponible au niveau de chaque instance.
onUninitialized, onLoading, onLoaded, onInteractive, onComplete	undefined	Fonctions de rappel pour les états XHR « fraîchement créé », « en cours d'envoi », « envoyé », « en réception » et « réception terminée », respectivement. Cela permet de séparer le code affecté à chaque étape du traitement, plutôt que d'avoir à tester la propriété <code>readyState</code> du requêteur. On n'utilise toutefois que rarement les deux premières.
onSuccess, onFailure, onXYZ	undefined	Il est possible de distinguer encore plus avant en séparant le code de traitement après requête couronnée de succès, de celui de réaction en cas de problème. On peut même associer un rappel à un code de retour précis (par exemple, 302, 201, etc. Il suffit de définir une association nommée d'après le code, comme <code>on302</code>). Ces fonctions sont appelées avant (et non à la place de) une éventuelle fonction associée à <code>onComplete</code> .
onException	undefined	Gestionnaire d'exceptions dans la couche client de requêtage. Prend deux arguments : l'objet <code>Ajax.Request</code> (ou <code>Ajax.Updater</code>) qui a rencontré un problème, et l'exception JavaScript qui a été levée.
parameters	''	Paramètres pour une requête GET (sans le ? initial). Il s'agit soit d'une chaîne URL-encodée, soit d'un tableau associatif (e.g. { name: 'Mark', age: 28 }), soit d'un Hash.
postBody	undefined	Corps de requête pour une requête POST. Ce sera généralement de l'URL encodé aussi, mais on préfère souvent passer par <code>parameters</code> , qui fonctionne aussi en post...
requestHeaders	undefined	Permet de définir des en-têtes de requête, par exemple un en-tête <code>Accept</code> pour demander un type de contenu réponse particulier. Il doit s'agir d'un tableau associatif, mais par souci de compatibilité on autorise encore un tableau classique au nombre d'éléments pair, constitué de couples nom/valeur. Par exemple, pour favoriser un résultat XML, on pourrait faire :

Tableau 7-2 Les options disponibles pour les requêteurs Ajax (suite)

Option	Défaut	Description
		<pre>new Ajax.Request('/mon_url_cible', { ... requestHeaders: { Accept: => 'application/xml' } ... });</pre> <p>Notez que Prototype définit par défaut les en-têtes personnalisés de réponse suivants :</p> <ul style="list-style-type: none"> • X-Request-With : 'XMLHttpRequest' • X-Prototype-Version : la version utilisée de Prototype, e.g. '1.6.0.3' • Accept : 'text/javascript, text/html, application/xml, */*'
sanitizeJSON	false	Détermine si une réponse JSON doit être systématiquement vérifiée syntaxiquement (méthode <code>isJSON()</code> de <code>String</code>) avant évaluation.

Quelques mots sur les fonctions de rappel

Les rappels `onUninitialized`, `onLoading`, `onLoaded` et `onInteractive` ne sont *pas garantis* : d'un navigateur à l'autre, ils ne se déclencheront pas toujours, ou pas toujours autant de fois, ou pas toujours dans l'ordre nominal ! Je vous conseille donc plutôt de les laisser de côté...

Les fonctions de rappel ne sont par ailleurs *pas bindées* automatiquement : elles utilisent le *binding* que vous leur fournissez, donc par défaut l'objet `window`.

Ceci étant, puisqu'elles prennent en argument un `Ajax.Response`, lequel fournit entre autres une propriété `request` référençant le requêteur courant, vous n'aurez besoin de définir le *binding* que si vous êtes déjà dans un objet et qu'une de ses méthodes utilisant `this` est passée comme fonction de rappel.

L'ensemble des fonctions de rappel (listées dans le tableau 7-2 par ordre chronologique) prennent deux arguments : un objet `Ajax.Response` représentant l'état actuel de la réponse, et l'état de l'en-tête `X-JSON` (qui ne sont donc intéressants qu'à partir du rappel `onInteractive`).

Ce deuxième argument n'est là que pour des raisons de compatibilité avec les anciennes versions, car `Ajax.Response` le fournit de manière plus cohérente désormais (voir la prochaine section). Déjà que l'en-tête en question est très peu utilisé...

Petite exception à la règle, le rappel `onException` justement, qui prend deux arguments autres : l'objet requêteur et l'exception JavaScript.

Interpréter la réponse avec Ajax.Response

Depuis Prototype 1.6 et grâce à Tobie Langel, le premier argument passé à toutes les fonctions de rappel est un objet `Ajax.Response`. Il représente du mieux possible l'état de la réponse au fil de la réception de celle-ci.

Il expose les propriétés suivantes :

Tableau 7–3 Les propriétés d'un objet `Ajax.Response`

Méthode	Description
<code>headerJSON</code>	Défini une fois les en-têtes de réponse reçus, il vaut <code>null</code> si aucun en-tête X-JSON n'a été reçu, et la valeur de l'en-tête autrement (mêmes règles d'évaluation que celles décrites pour <code>responseJSON</code>). À présent qu'on sait récupérer des réponses JSON complètes, on abandonne peu à peu cet en-tête qui, après tout, est limité en taille comme tous les en-têtes HTTP. Il n'est potentiellement utile qu'en accompagnement d'une réponse HTML, pour fournir une petite information complémentaire à la page.
<code>request</code>	L'objet requêteur (<code>Ajax.Request</code> ou <code>Ajax.Updater</code>) dont nous sommes en train d'étudier la réponse. Ne pas confondre avec <code>transport</code> .
<code>readyState</code>	Identique à la propriété native de <code>XMLHttpRequest</code> .
<code>responseJSON</code>	Le corps de réponse JSON évalué. N'est défini qu'une fois la réponse complètement reçue, à condition qu'elle ne soit pas « vide » (uniquement du whitespace), que le type MIME soit celui de JSON (<code>application/json</code>) et, si l'option <code>sanitizeJSON</code> est à <code>true</code> ou que la requête est inter-domaines, que son contenu soit syntaxiquement du JSON valide. Dans tous les autres cas, vaudra <code>null</code> .
<code>responseText</code>	Le corps de réponse brut, défini dès que celui-ci commence à être reçu. Démarrer à ' ', n'est donc jamais <code>null</code> .
<code>responseXML</code>	Le corps de réponse XML évalué. N'est défini qu'une fois la réponse complètement reçue, à condition que le type MIME s'y prête (généralement, type finissant en <code>/xml</code> ou <code>+xml</code>). Sinon, vaudra <code>null</code> .
<code>status</code>	Le code de réponse HTTP, par exemple 200, 302, 402...
<code>statusText</code>	Le texte complet du code de réponse HTTP, par exemple « 200 OK ».
<code>transport</code>	L'objet <code>XMLHttpRequest</code> sous-jacent.

On y trouve par ailleurs les méthodes suivantes :

Tableau 7–4 Les méthodes d'un objet `Ajax.Response`

Méthode	Description
<code>getAllHeaders()</code>	Renvoie tous les en-têtes de réponse une fois ceux-ci reçus ; en cas d'erreur, renvoie <code>null</code> .
<code>getAllResponseHeaders()</code>	Renvoie tous les en-têtes de réponse, ou lève une exception.

Tableau 7-4 Les méthodes d'un objet Ajax.Response (suite)

Méthode	Description
getHeader(nom)	Renvoie l'en-tête de réponse demandé une fois les en-têtes reçus ; en cas d'erreur ou d'en-tête manquant, renvoie null.
getResponseHeader(nom)	Renvoie l'en-tête de réponse demandé, ou lève une exception.

Lorsqu'une réponse est de type JavaScript (types MIME de partie primaire `text` ou `application` et de partie secondaire `javascript` ou `ecmascript`, éventuellement précédés de `x-`), elle sera automatiquement évaluée (à moins que l'option `evalJS` soit à `false`).

Pour les curieux, sachez que cette évaluation a lieu après les éventuels `onSuccess`, `onFailure` ou `onXYZ`, mais avant l'éventuel `onComplete`.

C'est bien pratique lorsque vous appelez une action serveur qui ne renvoie que du JavaScript : aucun traitement de réponse manuel à écrire !

Ajax.Updater : mettre à jour un fragment XHTML, exécuter un script

Évidemment, Prototype ne s'arrête pas là (ce n'est pourtant déjà pas mal !). Il fournit d'autres mécanismes, dédiés à des utilisations fréquentes d'Ajax. L'un d'eux est fourni par la classe `Ajax.Updater`, qui est spécialisée dans l'envoi de requêtes dont la réponse est censée mettre la page à jour. Il s'agit naturellement d'une classe fille de `Ajax.Request`.

Néanmoins, le constructeur n'est plus tout à fait le même : avant les arguments d'URL et d'options, `Ajax.Updater` prend un premier argument désignant l'élément qui fera l'objet d'une mise à jour. Comme la plupart du temps avec Prototype, cet argument peut être une `String` contenant l'identifiant de l'élément, ou l'élément lui-même. On verra un peu plus tard qu'il existe aussi une autre forme, plus avancée, pour ce premier argument.

Remarquez également que dans la mesure où le traitement de la réponse est entièrement automatique, définir des fonctions de rappel telles que `onComplete` ou `onSuccess` n'a pas beaucoup d'intérêt... S'il ne s'agit alors que de traitements génériques à toutes vos requêtes Ajax (journalisation, etc.), il existe de meilleurs moyens, que nous verrons un peu plus loin.

Armés des connaissances acquises dans le précédent chapitre, nous savons que les deux grands cas de figure pour une réponse modifiant la page sont : XHTML et

JavaScript. Il y a donc déjà une distinction potentielle quant au type du contenu renvoyé.

Il y a aussi diverses possibilités sur le type de la mise à jour : s'agit-il de remplacer un fragment du document (une valeur de pourcentage par exemple, qui aura peut-être évolué), ou d'ajouter au document ? Et dans ce deuxième cas, où l'insertion doit-elle avoir lieu ?

Malgré l'apparente complexité engendrée par la combinaison de ces possibilités, Prototype nous permet de préciser très simplement nos besoins à l'aide de deux options supplémentaires.

Tableau 7–5 Options supplémentaires pour Ajax.Updater

Option	Défaut	Description
evalScripts	false ^a	<p>Active l'évaluation automatique de scripts (c'est-à-dire de code JavaScript).</p> <p>Là, vous êtes tout surpris, car je vous ai expliqué tout à l'heure que <code>Ajax.Request</code> (et donc <code>Ajax.Updater</code>) évaluent automatiquement les scripts JavaScript renvoyés. C'est juste, mais uniquement si la réponse affiche un type MIME <code>JavaScript</code>.</p> <p>Afin de permettre à ceux qui ne pourraient pas manipuler les en-têtes dans la réponse HTTP de bénéficier de cette possibilité, mais aussi de pouvoir renvoyer des fragments HTML avec le script associé, Prototype offre cette option, dont l'activation exploitera les portions <code><script>...</script></code> du corps de la réponse.</p> <p>Notez au passage qu'en laissant cette option désactivée, Prototype vous protège contre des attaques malicieuses à base de script dans le contenu renvoyé (par exemple dans le cadre d'une syndication de contenu).</p>
insertion	undefined	Toute la flexibilité de <code>Ajax.Updater</code> vient de là ! En laissant cette option indéfinie, vous causerez le remplacement du contenu de l'élément désigné <i>via</i> le constructeur. C'est idéal pour mettre à jour un texte ou même un fragment DOM à remplacer complètement, par exemple tout ou partie d'un graphique SVG (ce qui est assez glamour, je trouve). Mais que faire lorsqu'il s'agit de compléter le document, par exemple pour ajouter un commentaire ou un envoi sur un forum en ligne « live » ? Il suffit de préciser ici l'une des classes d'insertion fournies par Prototype, c'est-à-dire l'une des positions d'insertion proposées par la méthode <code>insert</code> de <code>Element</code> . Vous en retrouverez la liste dans le tableau suivant.

a. En réalité, `undefined`, ce qui est plus cohérent pour marquer l'absence de définition. Mais comme vous le savez, évalué comme un booléen, cela équivaut à `false`.

Nous évoquons dans ce tableau le mécanisme des insertions, qui permet de ne pas remplacer le contenu de l'élément, mais plutôt d'ajouter du contenu. Ces ajouts dépendent de la position utilisée pour l'insertion et Prototype fournit des valeurs

adaptées à chaque besoin. Nous les avons déjà étudiées au chapitre 4, mais je vous offre un petit rappel. Les résultats présentés dans le tableau ci-dessous supposent le fragment XHTML initial suivant :

```
<p>
  <span id="colmin">Graff</span>
</p>
```

Et l'exécution ultérieure d'un code de type :

```
$('colmin').insert(position, ' <em>Hyrum</em> ')
```

Voici les positions d'insertion disponibles.

Tableau 7–6 Positions d'insertion pour l'ajout de contenu au DOM

Classe	Emplacement de l'insertion	Résultat
after	Immédiatement après l'élément.	<p> Graff Hyrum </p>
before	Immédiatement avant l'élément.	<p> Hyrum Graff </p>
bottom	Après le contenu de l'élément.	<p> Graff Hyrum </p>
top	Avant le contenu de l'élément.	<p> Hyrum Graff </p>

À présent que nous avons couvert à nouveau la théorie, voyons comment adapter notre exemple de réponse XHTML. Copiez votre répertoire de travail pour l'exemple de réponse XHTML et modifiez le fichier `client.js` : retirez notre fonction `getRequester` et modifiez `switchToAjax` pour obtenir le code suivant.

Listing 7–2 Le nouveau script client pour notre mise à jour XHTML

```
function bindForm() {
  $('#commentForm').observe('submit', switchToAjax);
} // bindForm

function switchToAjax(e) {
  e.stop();
```

```
new Ajax.Updater('comments', this.action,
{
  parameters: this.serialize(),
  insertion: 'bottom'
});
} // switchToAjax

document.observe('dom:loaded', bindForm);
```

Avouez que c'est impressionnant ! Vous pouvez lancer votre script `serveur.rb`, afficher `http://localhost:8042` et rafraîchir en forçant le contournement du cache, vous verrez : ça marche toujours (heureusement, ceci dit !). C'est « juste » plus court et plus élégant.

L'option `evalScripts` permet simplement de fournir dans le fragment de document que vous renvoyez un ou plusieurs scripts dédiés (éléments `<script>`), qui seront évalués immédiatement après l'insertion du fragment dans le DOM. Les éléments `<script>` eux-mêmes ne seront pas insérés dans le DOM.

Ce type de contenu mixte permet d'éviter de coupler votre fichier JavaScript à l'ensemble des réponses qu'il pourrait traiter, en plaçant les fragments de script associés à chaque réponse directement dans la réponse elle-même. Par exemple, une requête Ajax pourrait renvoyer ceci :

```
<li id="todo42">Aller chercher le pain</li>
<script type="text/javascript">
$( 'todo42' ).highlight();
</script>
```

Différencier la mise à jour entre succès et échec

Lorsque j'ai présenté `Ajax.Updater`, j'ai précisé qu'il existait une forme plus avancée pour le premier argument du constructeur. En effet, tel que nous l'avons utilisé jusqu'à présent, il effectue la mise à jour même si la requête a échoué (c'est-à-dire si, dans la fonction de rappel `onComplete`, la méthode `success()` renvoie `false`) ! Ce n'est pas forcément ce qu'on veut.

On a deux comportements alternatifs possibles :

- On souhaite une mise à jour en cas de réussite uniquement.
- On souhaite des mises à jour différentes suivant que la requête a réussi ou échoué (par exemple, on a un fragment de page dédié aux messages d'erreur, message que nous renverrait la couche serveur en cas de problème).

Ces deux cas de figure ont recours au même mécanisme : il s'agit de ne pas passer, comme premier argument au constructeur, la désignation d'un seul élément, mais plutôt de lui passer un objet désignant les éléments en cas de succès et d'échec.

Cet objet, généralement anonyme, a simplement besoin de fournir une désignation (l'élément lui-même ou une String avec son ID) dans sa propriété `success`, et éventuellement une seconde dans sa propriété `failure`.

- Si vous ne fournissez qu'une propriété `success`, l'échec ne donnera lieu à aucune mise à jour.
- Si vous fournissez les deux, l'échec mettra à jour l'élément désigné par la propriété `failure`.
- Il ne sert à rien (et il serait par ailleurs illogique) de ne préciser qu'une propriété `failure`...

Voici un exemple classique :

```
new Ajax.Updater({ success: 'cartItems', failure: 'errorMsg' },
  '/purchase',
{
  parameters: lineItemData,
  insertion: 'bottom'
});
```

Presque magique : Ajax.PeriodicalUpdater

Si `Ajax.Updater` vous a plu, vous allez adorer `Ajax.PeriodicalUpdater` ! Attention, la seconde n'est pas une classe fille de la première. En effet, il ne s'agit pas d'un type spécialisé de requêteur, mais d'un mécanisme de répétition intelligent autour d'une utilisation normale de `Ajax.Updater`.

Un objet `Ajax.PeriodicalUpdater` se construit strictement comme un objet `Ajax.Updater`, mais dispose de deux options supplémentaires.

Tableau 7-7 Options supplémentaires pour `Ajax.PeriodicalUpdater`

Option	Défaut	Description
<code>frequency</code>	2	Période (oui, Sam Stephenson n'est pas très au point sur le couple fréquence/période...), en secondes, entre deux invocations de <code>Ajax.Updater</code> . Évitez de préciser de trop petites valeurs (par exemple 0.1 alors que vous n'êtes pas en intranet et sur un serveur rapide), sans quoi vous risqueriez, au mieux, d'obtenir un comportement incohérent, au pire, de geler le navigateur, voire la machine.
<code>decay</code>	1	(Prononcez « di-kay ») Signifie littéralement <i>décomposition, décrépitude, déliquescence...</i> Et pourtant, c'est une fonctionnalité qui, pour être avancée, n'en est pas moins précieuse. Elle permet de demander un allongement progressif de la période (donc, pour les pointilleux, une baisse de la fréquence) lorsque le résultat ne varie pas d'une requête à l'autre. Explications plus en détail à la prochaine section.

Autre précision de taille : une méthode `stop()` permet d'arrêter le cycle d'invocations. S'il s'agit d'un arrêt temporaire, vous pouvez relancer le traitement ultérieurement avec la méthode `start()`, appelée automatiquement après l'initialisation de l'objet.

Par ailleurs, la fonction de rappel `onComplete` a ici une utilité, contrairement au cas de `Ajax.Updater`. En effet, la fonction de rappel serait invoquée à la fin de la méthode `stop()` (et non en fin de réception de réponse Ajax), et prendrait comme arguments tous ceux que vous auriez éventuellement passés à `stop()`.

Comprendre l'option `decay`

Voici comment l'option `decay` fonctionne : à chaque requête dont le résultat est identique à celui de la précédente, la valeur active de `decay` (qui démarre à celle de l'option) est multipliée par la valeur de l'option. La période avant la prochaine requête est alors définie en multipliant la période (option `frequency`) par le `decay` actif. En revanche, dès qu'une requête ramène un résultat différent, le `decay` actif repasse à 1 (un).

Donc, une option `decay` de 1 (un) revient à désactiver cette fonctionnalité. C'est le cas par défaut. Et une option `decay` inférieure à un serait contre-productive : alors que rien n'a l'air de se passer côté serveur, on demanderait de plus en plus souvent ce qui se passe !

Imaginons en revanche une option `decay` à 2 (histoire de simplifier les valeurs, parce qu'en réalité, on opte plutôt pour une valeur entre 1,1 et 1,5 dans la plupart des cas). Voici un petit tableau résumant une séquence d'appel.

Tableau 7-8 Exécution d'un `Ajax.PeriodicalUpdater`, option `decay` à 2

Moment	Decay actif	Période	transport. responseText	Réaction
00:00	2	2	42	Réponse différente (pas d'antérieur) : <code>decay</code> 1.
00:02	1	2	54	Idem.
00:04	1	2	54	Réponse identique : <code>decay</code> × option <code>decay</code> , et comme période = option <code>frequency</code> × <code>decay</code> ...
00:08	2	4	57	Réponse différente : <code>decay</code> 1.
00:10	1	2	57	Même comportement qu'à 00:04.
00:14	2	4	57	C'est le début du grand ralentissement...
00:22	4	8	57	...
00:38	8	16	57	32 secondes avant la prochaine requête !
01:10	16	32	63	Ah, enfin ! On retombe à <code>decay</code> 1.
01:12	1	2	64	...
01:14	1	2	66	Et visiblement, le serveur s'est « décoincé » !

Petits secrets supplémentaires

Tant qu'à explorer les possibilités Ajax de Prototype, autant ne pas se limiter aux utilisations courantes (je parie que vous aussi, vous aimez en savoir plus que les autres), non ? Jetons donc un coup d'œil sur quelques points moins fréquemment mis en lumière.

Commençons avec deux petits détails : l'espace de noms Ajax fournit, en plus des trois classes que nous venons d'explorer, une méthode `getTransport()` et une propriété `activeRequestCount`.

La première est sans mystère : `Ajax.getTransport()` est la méthode appelée, en interne, par les classes de requête pour obtenir le requêteur, c'est-à-dire l'objet `XMLHttpRequest` à proprement parler. C'est l'équivalent de notre bonne vieille fonction `getRequester`.

La propriété `Ajax.activeRequestCount` est tenue à jour automatiquement par Prototype et fournit le nombre de requêteurs créés mais dont le traitement n'est pas encore terminé. On imagine l'utilité éventuelle de cette propriété dans un contexte de journalisation ou de surveillance de l'activité Ajax de la page.

Et cependant...

Puisque `getTransport()` se contente de renvoyer un objet `XMLHttpRequest`, objet sur lequel il n'a pas de contrôle particulier, comment diable Prototype fait-il pour être tenu au courant de la fin de traitement des requêteurs ? (Vous vous êtes bien posé la question, n'est-ce pas ?)

Si vous êtes comme moi, vous n'aimez pas l'impression de boîte noire qu'on a en découvrant un nouveau framework. Dissipons donc rapidement le voile de mystère qui plane sur cet apparent tour de force : Prototype utilise tout simplement le dernier secret dont je voulais vous parler : `Ajax.Responders`.

`Ajax.Responders` est un objet global qui fournit un point centralisé d'inscription pour des fonctions de rappel globales à tous les requêteurs. En d'autres termes, une fonction de rappel enregistrée auprès de `Ajax.Responders` s'ajoute à celles définies pour tout `Ajax.Request` ou `Ajax.Updater` (et par conséquent, tout `Ajax.PeriodicalUpdater` également).

Ce qui signifie, incidemment, que si vous obtenez un requêteur manuellement en appelant `getTransport`, il n'est pas concerné (remarquez bien que vous n'avez pas de raison particulière de bouder ainsi les mécanismes standards).

Prototype se contente donc d'enregistrer d'office auprès de `Ajax.Responders` deux fonctions de rappel, pour `onCreate` et `onComplete`, qui maintiennent notre fameux `activeRequestCount` à jour. C'est tout simple !

En ce qui nous concerne, `Ajax.Responders` a deux méthodes intéressantes : `register` et `unregister`, dont les noms sont plutôt explicites. Chacune prend en paramètre un

objet dont les méthodes ont les noms des rappels souhaités. Il peut s'agir d'objets anonymes, comme justement dans Prototype :

```
Ajax.Responders.register({
  onCreate: function() {
    Ajax.activeRequestCount++;
  },
  onComplete: function() {
    Ajax.activeRequestCount--;
  }
});
```

Les fonctions de rappel globales ainsi définies sont appelées immédiatement après les fonctions de rappel spécifiques à chaque requêteur (telles que celles que nous avions définies jusqu'à présent). Elles prennent toutefois un premier argument avant les deux usuels : le requêteur concerné.

Notez toutefois que les rappels spéciaux `onSuccess`, `onFailure` et `onXYZ` (rappels numériques) ne participent pas à ce mécanisme : il ne peut pas y avoir de tels rappels globaux. Seuls les rappels calés sur le cycle de vie d'un requêteur, ainsi que les gestionnaires d'exceptions, sont éligibles à un traitement global.

Notez aussi que les rappels globaux sont appelés dans le contexte de l'objet que vous aurez passé à `register`, ce qui permet de passer des objets complexes avec d'autres propriétés et méthodes et d'utiliser `this` dans vos rappels pour y accéder.

Enfin, pour la petite histoire, sachez que `Ajax.Responders` est un `Enumerable`. Si cela ne vous dit rien, prêtez donc l'oreille : entendez-vous le champ de sirène du chapitre 4 ? Ne lui résistez pas ! Je vous attendrai sagement ici.

Un dernier secret pour la route, que j'avais annoncé dans le chapitre 4. Il existe un cas de figure très courant, qui consiste simplement à prendre un formulaire et à le « court-circuiter vers Ajax », sans que la réponse Ajax ne nécessite une insertion explicite dans le DOM (en somme, en utilisant un `Ajax.Request` et non un `Ajax.Updater`). Notez que cela n'exclut pas toute forme d'insertion : on peut très bien renvoyer une réponse de type JavaScript qui contienne du code faisant des insertions DOM...

Ce court-circuit revient généralement à procéder de la façon suivante :

- On utilise l'attribut `method=` du formulaire pour l'option `method`.
- On utilise l'attribut `action=` du formulaire pour l'argument d'URL.
- On sérialise le formulaire pour l'option `parameters`.

Eh bien c'est exactement ce que fait la méthode `request` ajoutée par Prototype aux formulaires. Elle prend simplement un tableau associatif d'options en paramètre,

lesquelles options vont remplacer ou compléter les options définies par défaut. Un court-circuit Ajax peut donc être, potentiellement, aussi court que ceci :

```
$(‘myForm’).observe(‘submit’, function(e) {  
    e.stop();  
    this.request();  
});
```

Voilà, croyez-le ou non, entre le chapitre 4 et celui-ci, nous avons fait une exploration totale de Prototype.

Voici donc une des promesses de ce livre, et plus particulièrement de ce chapitre, qui est tenue. Mais je ne vous ai pas appâté jusqu’ici sur la base de cette seule promesse, j’ai aussi fait allusion au monde merveilleux des effets visuels riches et d’interfaces utilisateurs très dynamiques, avec du glisser-déplacer, de la complétion automatique de texte, et autres merveilles... Il est temps d’assumer ces allusions. Le chapitre 8 va vous plonger dans *script.aculo.us*.

Pour aller plus loin...

Livres

Prototype and script.aculo.us: you never knew JavaScript could do this!

Christophe Porteneuve

Pragmatic Programmers, décembre 2007, 425 pages

ISBN 1-934356-01-8

Practical Prototype and script.aculo.us

Andrew Dupont

Apress, juin 2008, 328 pages

ISBN 1-59059-919-5

Sites

On compte aujourd’hui de nombreuses communautés Ajax établies. Pour n’en citer que trois :

- **Ajaxian** : <http://ajaxian.com>
- **AjaxPatterns** : <http://ajaxpatterns.org>
- **Ajax Developer’s Journal** : <http://ajaxdevelopersjournal.com>

Baekdal.com, le site de Thomas Baekdal. Vous y trouverez de nombreux articles de bonne qualité (voire excellents) autour d’Ajax et des technologies Web 2.0 : <http://baekdal.com>.

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages offerts par les outils d'indexation et de recherche de Google. On y trouve de nombreux membres ayant un bon niveau technique. Hélas, il a fini par crouler sous le *spam* faute d'une modération active.

<http://groups.google.com/group/rubyonrails-spinoffs>

Heureusement, à l'initiative de TJ. Crowder, un groupe de remplacement, au nom d'ailleurs plus explicite, a vu le jour en juillet 2008 et concentre désormais toute l'activité, le groupe historique n'existant plus que pour ses archives :

<http://groups.google.com/group/prototype-scriptaculous>.

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse rapide et fiable sur le canal IRC dédié à Prototype, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#prototype`.

8

Une ergonomie de rêve avec script.aculo.us

Il est temps d'explorer l'univers des effets visuels, des comportements avancés, et de leur impact ergonomique avec script.aculo.us. Nous prendrons aussi le temps de réfléchir à la problématique, parfois subtile, de l'accessibilité des interfaces utilisant Ajax.

Pour des raisons de taille, ce chapitre ne couvre toutefois pas la totalité de script.aculo.us 1.8.1 ; il se concentre sur les parties les plus utilisées : les effets visuels, le glisser-déplacer, le tri et la complétion automatique. On laissera en revanche de côté la modification en place, la construction de fragments DOM et la lecture de sons. La bibliographie en fin de chapitre vous donne des références complètes à consulter si vous en éprouvez le besoin.

Une ergonomie haut de gamme avec script.aculo.us

Rendons visite à Thomas Fuchs, le jeune Autrichien auteur de script.aculo.us (entre autres jolies choses).

Cette bibliothèque JavaScript réussit le véritable tour de force de mettre à disposition de tout un chacun, sous forme de petits objets JavaScript faciles d'emploi, des effets visuels et éléments d'interfaces utilisateur complexes, et cela, sur la plupart des navigateurs (comprendre : sur MSIE aussi !). Chapeau bas, Herr Fuchs !

Mais je sens bien que je ne saurais vous convaincre sur la seule base de mon enthousiasme débordant. Dans ce cas, permettez-moi de vous montrer.

Commencez par récupérer script.aculo.us : vous la trouverez à la racine de l'archive des codes source, ou sur l'excellent site de la bibliothèque, rempli de documentations, exemples, démonstrations, etc. : <http://script.aculo.us>. Choisissez l'onglet `downloads` et prenez la version la plus récente (1.8.1 à l'heure où j'écris ces lignes), dans le format d'archive que vous préférez. Vous y piocherez les fichiers JS dont nous aurons besoin (pas tous !) au fil de nos exemples, dans les répertoires `lib` et `src` de l'archive.

QUALITÉ Des tests à foison !

script.aculo.us montre l'exemple en prouvant avec brio qu'il est parfaitement possible d'appliquer une méthodologie de test rigoureuse à du code JavaScript. La bibliothèque est dotée de plus de 80 tests unitaires regroupant environ 1 000 assertions, exécutables et consultables de façon automatisée. Plusieurs dizaines de tests fonctionnels sont également maintenus.

Ouvrez par exemple la page `test/run_unit_tests.html` dans votre navigateur et exécutez les tests unitaires. Vous allez être impressionnés !

Les effets visuels

Nous allons d'abord examiner les effets visuels. Ils sont découpés en deux catégories :

- Les effets dits « noyau » (*core effects*), qui sont fondamentaux.
- Les effets dits « combinés », qui associent des effets noyau et ajoutent parfois du comportement supplémentaire pour obtenir des effets avancés.

Avant de réaliser quelques exemples, nous allons explorer les effets noyau et les concepts généraux d'utilisation des effets. La prochaine section contient de nombreuses informations de référence.

Les effets noyau

Tous les effets, noyau comme combinés, sont fournis sous forme de classes disponibles dans l'objet global `Effect`. On compte 9 effets noyau dans `script.aculo.us`, sur lesquels se basent tous les autres. En voici une description rapide, mais nous verrons des exemples concrets plus loin dans ce chapitre.

Tableau 8–1 Effets noyau de `script.aculo.us`

Classe	Description
<code>Effect.Event</code>	Permet de placer un fragment JavaScript quelconque au sein d'une file d'effets. Nous verrons les files d'effets plus loin dans ce chapitre.
<code>Effect.Highlight</code>	Classiquement connu sous le nom <i>Fade To Yellow</i> , cet effet réalise un fondu enchaîné de couleurs de fond pour l'élément. Idéal pour attirer l'attention de l'utilisateur sur un élément fraîchement ajouté ou modifié par un traitement Ajax. Nous verrons plus bas quelques précautions d'emploi à observer.
<code>Effect.Morph</code>	C'est un des effets « couteau suisse » : il permet de faire évoluer des propriétés CSS d'un élément vers des valeurs cibles, exprimées individuellement ou au travers d'une classe CSS. Voir aussi <code>Effect.Transform</code> .
<code>Effect.Move</code>	Déplace l'élément d'un certain nombre de pixels, verticalement et horizontalement. Le déplacement est soit absolu (on fournit une position cible), soit relatif (on fournit les déplacements horizontal et vertical).
<code>Effect.Opacity</code>	Modifie l'opacité (ou la transparence, suivant le point de vue) de l'élément. Une opacité de 0 % est une transparence totale (position 0 . 0 de l'effet).
<code>Effect.Parallel</code>	Constitue la base de réalisation des effets combinés, en permettant de synchroniser plusieurs effets (noyau ou combinés), généralement sur un même élément. Nous verrons dans un exemple que sa syntaxe d'invocation est particulière.
<code>Effect.Scale</code>	Ajuste progressivement la taille de l'élément et de son contenu jusqu'à un pourcentage donné. Dispose de nombreuses options spécifiques.
<code>Effect.Transform</code>	Potentiellement, l'effet bulldozer : il permet de définir une série de transformations à la Morph sur des sélections d'éléments, et de réaliser les transformations correspondantes à la demande. Assez impressionnant !
<code>Effect.Tween</code>	Modifie un objet quelconque sur la base d'une séquence de valeurs (les positions de l'effet) qui sont soit affectées à une propriété de l'objet, soit passées à une méthode de l'objet.

Invocation de l'effet

Lancer un effet noyau fait toujours appel à la même syntaxe (sauf pour un `Parallel`) :

SYNTAXE

```
new Effect.NomDeLEffet(element[, parametres_requis][, options]);
eltÉtendu.highlight([options]) -> eltÉtendu
eltÉtendu.morph(style[, options]) -> eltÉtendu
eltÉtendu.visualEffect(nomEffet[, options]) -> eltÉtendu
```

Comme d'habitude, le premier argument peut être un identifiant ou l'élément lui-même. Tant les paramètres que les options sont conceptuellement des *hashes*, donc la plupart du temps des objets anonymes, avec une propriété par paramètre ou option. Enfin, notez le `new` au début, qui est trop souvent omis, mais devient vite obligatoire lorsqu'on souhaite déclencher plusieurs effets en parallèle.

Les paramètres requis varient d'un effet à l'autre ; dans certains cas, il n'y a aucun paramètre requis : on peut alors s'en passer et ne fournir que d'éventuelles options.

Les éléments étendus bénéficient par ailleurs de méthodes pour certains effets noyau (`highlight` et `morph`) et pour tous les effets combinés. Une méthode générique `visualEffect` fournit une syntaxe d'appel alternative, quel que soit l'effet ciblé.

Options communes à tous les effets noyau

Tout effet noyau reconnaît les options suivantes, qu'on retrouve dans `Effect.DefaultOptions` :

Tableau 8–2 Options communes à tous les effets noyau

Option	Description
<code>delay</code>	Attente, en secondes, avant le démarrage de l'effet. <code>0.0</code> par défaut.
<code>duration</code>	Durée de l'effet, en secondes. Nombre à virgule. <code>1.0</code> par défaut.
<code>fps</code>	Nombre d'ajustements par seconde (<i>frames per second</i>). <code>100</code> par défaut, très largement suffisant, qui correspond sur la plupart des matériels récents à <code>66 frames</code> réels. Mais je ne vois vraiment pas quel intérêt on aurait à dépasser environ <code>50</code> , seuil absolu de la vision humaine, et <code>25</code> est également bien fluide.
<code>from</code>	Position au sein de l'effet en début de traitement. Nombre entre <code>0.0</code> (0 %) et <code>1.0</code> (100 %). <code>0.0</code> par défaut. Certains effets particuliers en changent la sémantique, par exemple <code>Tween</code> . Voir aussi l'option <code>to</code> .
<code>queue</code>	Permet de placer l'effet courant dans une file d'effets. Peut être un nom de file (' <code>parallel</code> ' par défaut) ou un objet spécial. Voir la section dédiée aux files d'effets plus loin dans ce chapitre.
<code>sync</code>	Détermine si la progression est automatique (<code>true</code> , par défaut), ou s'il faut manuellement progresser (<code>false</code>) en appelant <code>render()</code> sur l'effet. Utilisé en particulier dans le cadre d'un <code>Effect.Parallel</code> pour gérer plusieurs effets s'exécutant de façon synchronisée.
<code>to</code>	Option symétrique à <code>from</code> , pour la fin du traitement. <code>1.0</code> par défaut.

Tableau 8–2 Options communes à tous les effets noyau (suite)

Option	Description
transition	Fonction assurant la progression de l'effet dans le temps (succession des valeurs entre from et to). Il existe 9 transitions prédéfinies, toutes dans Effect.Transitions : – <code>sinoidal</code> (par défaut) accélère l'effet peu après le démarrage et le ralentit peu avant la fin. C'est le comportement le plus « naturel » ; – <code>spring</code> , la plus récente, réalise un <i>overshoot</i> : un léger dépassement de la position finale pour finalement se stabiliser sur cette dernière, un peu comme un ressort qu'on relâche après l'avoir tendu. Très sympathique dans certains cas, comme par exemple pour un lâcher d'abandon en cours de glisser-déplacer ; – <code>linear</code> maintient une vitesse fixe, souvent moins élégante ; – <code>reverse</code> inverse le déroulement de l'effet : il s'effectue de sa fin à son début. ; – <code>wobble</code> amène un comportement « dessert en gelée » à l'évolution de l'effet lui-même, très visible sur, par exemple, Effect.Scale ; – <code>flicker</code> donne une impression de clignotement basée sur les 25 % finaux de l'effet ; – <code>pulse</code> accélère un effet de base pour le répéter un certain nombre de fois (5 par défaut) au cours de sa durée prévue (utilisé par exemple pour réaliser l'effet Pulse) ; – <code>none</code> laisse l'élément dans son état en début d'effet, ce qui permet de désactiver l'effet pendant un débogage sans le retirer pour autant du code source ; – <code>full</code> amène immédiatement l'élément en fin d'effet. Même remarque que pour none.

Fonctions de rappel

En plus des options à proprement parler, le troisième argument du constructeur peut également être utilisé pour associer des fonctions de rappel à l'effet (ce dont nous avons rarement besoin nous-même hormis pour afterFinish, mais qui est fort utile pour la création d'effets combinés). Les points de rappel disponibles sont, par ordre chronologique : beforeSetup, afterSetup, beforeStart, beforeUpdate, afterUpdate et afterFinish. Les fonctions sont appelées avec l'objet Effect en argument.

Lorsqu'on réalise ses propres effets, on utilise en général les variantes à suffixe Internal de ces rappels, afin de laisser les rappels « publics » au code utilisateur.

Et si on essayait quelques effets ?

Mais bien sûr ! D'ailleurs, nous n'aurons même pas besoin d'une couche serveur pour nos tests, tout peut se faire côté client, en JavaScript, avec un doigt de XHTML quand même...

Créez un répertoire de travail, par exemple `script.aculo.us`, ainsi qu'un premier sous-répertoire `opacity`, et placez à l'intérieur, en plus de `prototype.js` et du fichier `effects.js` de `script.aculo.us`, les fichiers suivants, que nous adapterons pour nos essais ultérieurs.

Listing 8-1 Notre index.html pour tester Effect.Opacity

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Test de Effect.Opacity</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>

<body>

<h1>Test de <code>Effect.Opacity</code></h1>

<p id="byebye">Cette page teste l'effet Effect.Opacity. Cliquez sur ce
paragraphe pour avoir une démonstration.</p>

</body>
</html>
```

Listing 8-2 Notre tests.js pour tester Effect.Opacity

```
function applyEffect(e) {
  e.stop();
  new Effect.Opacity('byebye', { duration: 2, from: 1, to: 0 });
} // applyEffect

function bindTestElements() {
  $('#byebye').observe('click', applyEffect);
} // bindTestElements

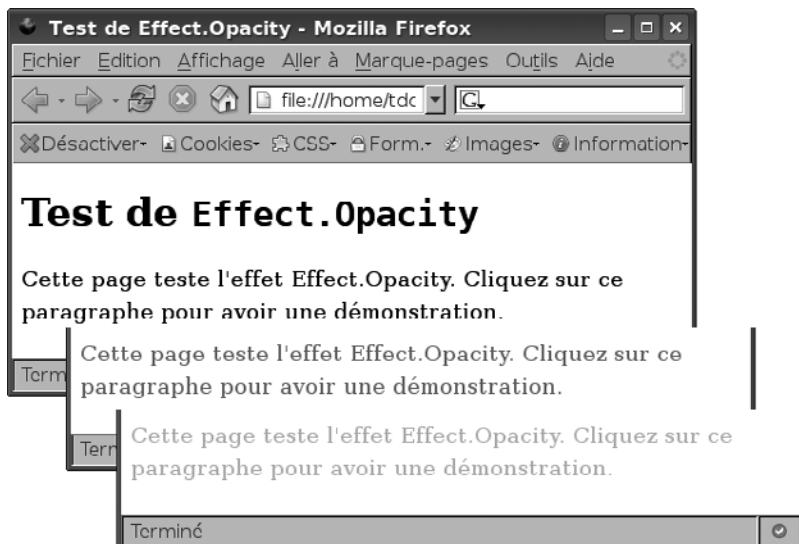
document.observe('dom:loaded', bindTestElements);
```

Listing 8-3 Notre feuille de styles, toute simple

```
#byebye {
  font-size: large;
  width: 60ex;
  line-height: 1.5em;
  cursor: default;
}
```

Notez que nous avons, volontairement, ralenti l'effet en calant sa durée sur 2 secondes au lieu d'une. Par ailleurs, puisqu'il s'agissait ici de réduire l'opacité, il a fallu inverser les valeurs par défaut des options `from` et `to`. Voyons le résultat en actionnant le bouton.

Figure 8–1
L'effet d'opacité en action



À présent, copions notre répertoire sous le nom `scale`, et voyons `Effect.Scale`, qui dispose de nombreuses options. Nous allons brièvement les passer en revue, puis en appliquer quelques-unes.

Tableau 8–3 Les options spécifiques à `Effect.Scale`

Option	Description
<code>scaleX</code>	Active le redimensionnement horizontal (<code>true</code> par défaut).
<code>scaleY</code>	Active le redimensionnement vertical (<code>true</code> par défaut).
<code>scaleContent</code>	Active le redimensionnement du contenu (<code>true</code> par défaut, fait évoluer <code>font-size</code>).
<code>scaleFromCenter</code>	Maintient la position du centre au fil du redimensionnement (<code>false</code> par défaut).
<code>scaleMode</code>	Trois possibilités : - 'box' (par défaut) : redimensionne la partie visible de l'élément. - 'content' : redimensionne la totalité de l'élément, en prenant en compte les parties non visibles (qui nécessitaient un défilement, etc.). - objet anonyme avec propriétés <code>originalWidth</code> et <code>originalHeight</code> , décrivant la taille de départ à utiliser.
<code>scaleFrom</code>	Indique le pourcentage de départ à utiliser (par rapport à la taille originale ; 100 par défaut).

Que d'options ! Je vous accorde que la différence entre les modes `box` et `content` est un peu obscure... Notez par ailleurs que la construction exige un paramètre (deuxième argument), qui est le pourcentage final de redimensionnement (ainsi, 50 aura réduit la taille par 2, 110 aura gagné 10 %).

Adaptions donc notre script pour réaliser un agrandissement horizontal centré à 150 %, qui ne touchera pas à la taille du contenu. Après avoir ajusté le HTML pour qu'il indique le bon effet, il suffit de modifier `applyEffect` comme suit :

Listing 8-4 Un exemple de redimensionnement

```
function applyEffect(e) {
    e.stop();
    new Effect.Scale('byebye', 150, {
        duration: 2, scaleY: false, scaleContent: false,
        scaleFromCenter: true
    });
} // applyEffect
```

Et nous allons ajuster la CSS pour que le résultat soit plus parlant.

Listing 8-5 Notre CSS ajustée pour renforcer l'impact visuel de l'effet

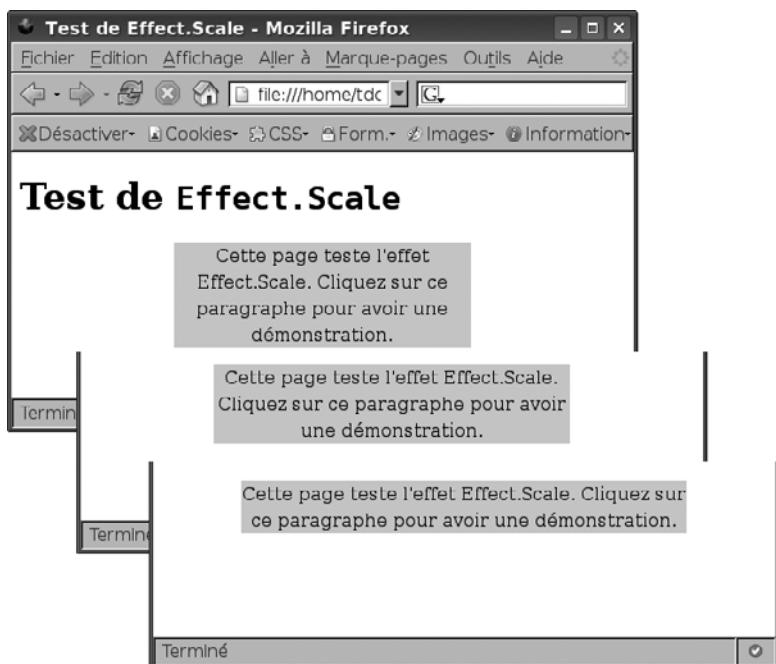
```
#byebye {
    margin: 1em auto 0;
    width: 30ex;
    font-size: 100%;
    line-height: 1.5em;
    text-align: center;
    cursor: default;
    background: #cd9;
}
```

Et voici le résultat !

Il faut bien comprendre d'où viennent les dimensions originales du paragraphe : la largeur est fixée par CSS (30ex), mais la hauteur est dynamique, simple fonction du nombre de lignes. C'est pourquoi, au fur et à mesure de l'élargissement du paragraphe, les lignes se font suffisamment longues pour être moins nombreuses, et la hauteur diminue (puisque l'effet, lui, n'y touche pas explicitement, conformément à notre option `scaleY: false`).

Cela ne se voit pas forcément à l'impression, mais en testant, vous remarquerez que le centre horizontal du paragraphe est fixe lors du redimensionnement : c'est le rôle de notre option `scaleFromCenter`.

Figure 8–2
L'effet Scale en action



Voyons à présent un exemple de `Highlight`. Cet effet est simple, mais il est très couramment utilisé pour mettre en exergue visuelle un élément de la page qui a changé ou vient d'apparaître (une pratique incontournable pour améliorer l'accessibilité en Ajax). L'effet a quatre options spécifiques.

Tableau 8–4 Les options spécifiques à Effect.Highlight

Option	Description
<code>startcolor</code>	Expression de couleur CSS indiquant la couleur de fond en début d'effet. Par défaut, '#fffff99', équivalent de '#ff9', soit un jaune clair.
<code>endcolor</code>	Même chose mais pour la couleur de fin. Par défaut la couleur de fond appliquée à l'élément (propriété CSS <code>background-color</code>), ou à défaut '#ffffff', donc blanc.
<code>restorecolor</code>	Définit la couleur de fond après l'effet. Indéfinie par défaut, elle amène script.aculo.us à tenter de récupérer la couleur de fond courante de l'élément (celle avant que l'effet ne démarre), ce qui n'est garanti pour tous les navigateurs que si celle-ci est indéfinie ou exprimée avec la syntaxe <code>rgb(rouge, vert, bleu)</code> .
<code>keepBackgroundImage</code>	Détermine si l'image de fond éventuelle est préservée lors du fondu de couleur (ce qui n'a d'intérêt que si l'image utilise une transparence (GIF) ou un canal alpha (PNG) pour que la couleur de fond soit visible « à travers »). <code>false</code> par défaut, ce qui retire l'image de fond pendant le fondu.

Copiez votre répertoire précédent dans un nouveau répertoire nommé `highlight`, ajustez à nouveau le HTML pour refléter l'effet, puis modifiez la fonction `applyEffect` du script comme suit.

Listing 8-6 Un exemple de mise en exergue (highlight)

```
function applyEffect(e) {
    e.stop();
    new Effect.Highlight('byebye', { duration: 2 });
} // applyEffect
```

La couleur de fond prédéfinie de notre paragraphe nous permet de voir ici la restauration de la couleur de fond originale : on a un fondu du jaune clair vers le vert clair, c'est impeccable !

Une impression noir et blanc ne donnerait rien de bien utile, aussi pour cette fois, on se passera d'une figure...

Passons à `Effect.Move`. Associé à `Effect.Scale`, il fournit une bonne partie des effets combinés... Il propose trois options spécifiques :

Tableau 8-5 Les options spécifiques à `Effect.Move`

Option	Description
x	La position horizontale cible, ou le déplacement horizontal. Zéro par défaut.
y	La position verticale cible, ou le déplacement vertical. Zéro par défaut.
mode	Détermine si l'on doit atteindre une position absolue (' <code>absolute</code> ') ou si l'on effectue plutôt un déplacement relatif (' <code>relative</code> ', valeur par défaut).

Copiez votre répertoire précédent dans un nouveau répertoire nommé `move`, ajustez à nouveau le HTML pour refléter l'effet, puis modifiez la fonction `applyEffect` du script pour proposer les deux modes de déplacement suivant qu'on aura enfoncé la touche *Alt* ou non, comme suit :

Listing 8-7 Un exemple de déplacements absolu et relatif

```
function applyEffect(e) {
    e.stop();
    new Effect.Move('byebye', {
        duration: 2, x: 150, y: 50,
        mode: (e.altKey ? 'absolute' : 'relative')
    });
} // applyEffect
```

Afin de mieux voir la notion d'absolu, nous allons faire en sorte que notre élément de référence soit positionné en absolu :

```
function bindTestElements() {  
    $('byebye').absolutize().observe('click', applyEffect);  
} // bindTestElements
```

Chargez la page et testez : cliquez une ou deux fois pour voir un déplacement relatif s'opérer, puis cliquez tout en maintenant la touche Alt enfoncee pour voir un déplacement vers les coordonnées absolues (150, 50), tout près du titre...

Voyons à présent `Effect.Tween`. Ce petit effet est parfois réutilisé en interne par `script.aculo.us`, par exemple pour implémenter `Effect.ScrollTo`. Il permet de faire évoluer une propriété d'un objet quelconque (pas nécessairement un élément du DOM !), soit en affectant directement une valeur à la propriété, soit en appelant une méthode. Les positions de l'effet, qui peuvent être n'importe quelle séquence de valeurs, sont utilisées comme valeur de la propriété ou argument de la méthode.

La syntaxe d'appel a deux arguments requis : `from` et `to`, qui donnent la séquence de valeurs :

SYNTAXE

```
new Effect.Tween(objet, from, to[, options], nomPropOuMéthode)
```

Copiez votre répertoire `highlight` dans un nouveau répertoire nommé `tween`, ajustez à nouveau le HTML pour refléter l'effet mais modifiez le corps comme ceci :

Listing 8-8 Notre HTML ajusté pour illustrer `Effect.Tween`

```
<p id="byebye">Cette page teste l'effet Effect.Tween.</p>  
  
<form>  
    <p><input type="button" id="btnTweenFont" value="Sur la taille  
        ↵ de fonte" /></p>  
    <p><input type="button" id="btnTweenColor" value="Sur la couleur  
        ↵ de fonte" /></p>  
    <p><input type="button" id="btnTweenOpacity" value="Sur l'opacité" />  
        ↵ </p>  
</form>
```

Nous allons modifier tests.js comme suit :

Listing 8-9 Le script de démonstration pour Effect.Tween

```
function applyEffect(e, prop, from, to) {
    e.stop();
    new Effect.Tween('byebye', from, to, { duration: 2 }, prop);
} // applyEffect

function bindTestElements() {
    $('#btnTweenOpacity').observe('click',
        applyEffect.bindAsEventListener(null, 'setOpacity', 1, 0.3));
    $('#btnTweenFont').observe('click',
        applyEffect.bindAsEventListener(null, function(p) {
            this.setStyle({ fontSize: p.toFixed(3) + 'em' });
        }, 1, 5));
    $('#btnTweenColor').observe('click',
        applyEffect.bindAsEventListener(null, function(p) {
            var colorString = '#' + p.ceil().toColorPart() + '00' +
                (p / 2).ceil().toColorPart();
            this.setStyle({ color: colorString });
        }, 0, 255));
} // bindTestElements
```

Nous avons là du code un peu avancé... Tout d'abord, remarquez qu'on utilise `bindAsEventListener`, pour pré-remplir des arguments de notre gestionnaire d'événements tout en lui garantissant l'objet événement en première position.

Ensuite, nous avons une première définition pour le bouton d'opacité, qui est assez simple : notre élément est doté d'une méthode `setOpacity` qu'on peut appeler directement, ici avec des valeurs normalisées allant de 1 (pleine opacité) à 0.3 (30 % d'opacité, ou 70 % de transparence si vous préférez).

Les deux définitions suivantes sont plus complexes. La première illustre une modification de style, qui exige de passer les bons arguments à `setStyle`, dont le seul nom ne suffirait pas... On veut faire évoluer la taille de fonte de `1em` à `5em`, mais `Tween` ne prend par ailleurs que des valeurs numériques : on va donc ajouter le suffixe « `em` » dans notre fonction.

Quant à la dernière fonction de rappel, elle compose dynamiquement une chaîne hexadécimale de code couleur déterminé d'après une valeur allant de 0 à 255, sachant qu'elle veut passer de `#000000` (le noir par défaut) à `#ff0088` (une sorte de fuschia, dont la composante bleu est la moitié de la composante rouge, ce qui simplifie nos calculs). C'est là que les extensions de Prototype à `Number`, comme `ceil()` (arrondi vers le haut) et `toColorPart()` (représentation hexadécimale sur deux chiffres) sont bien pratiques...

C'est le moment de charger la page et de jouer. Comme les positions de début du `Effect.Tween` sont explicites, vous pouvez les ré-essayer autant de fois que vous voulez sans recharger.

Essayez aussi de déclencher plusieurs évolutions en parallèle (ce sera plus facile avec celle de taille de fonte en dernier, car elle fait bouger les boutons vers le bas) !

`Effect.Tween` est merveilleusement versatile, mais on l'a vu, lorsqu'on souhaite faire évoluer des propriétés de style, il faut passer par une conversion manuelle au sein d'une fonction de rappel personnalisée. C'est un peu lourd, sans parler du fait qu'on peut vouloir changer plusieurs propriétés *en même temps*. Ce type de besoin est exactement ce à quoi répond `Effect.Morph`.

SYNTAXE

```
new Effect.Morph(element, { style: leStyle[, ...] })
eltÉtendu.morph(leStyle[, options]) -> eltÉtendu
```

`Effect.Morph` permet de définir un style cible, sous la forme d'une série de propriétés avec leurs valeurs finales. Cette série est exprimée soit sous forme textuelle (e.g. `'font-size: 5em; color: #f08'`), soit sous forme d'un tableau associatif ou d'un véritable Hash (e.g. `{ fontSize: '5em', color: '#f08' }`), soit sous forme d'une classe CSS (e.g. `'morphed'`).

L'effet va réaliser une évolution de l'ensemble des propriétés *interpolables*, ce qui se résume aux propriétés numériques (dont les tailles, bordures, marges et espacements, mais aussi l'opacité) et aux couleurs. À noter que pour les tailles de fonte, travailler en pixels plutôt qu'en unités relatives est, hélas, recommandé pour des raisons de portabilité entre navigateurs...

Des propriétés comme la graisse (`font-weight`), l'italique, le soulignement, la famille de police ou le type de bordure ne sont pas interpolables, et ces valeurs cibles ne deviendront donc actives que brusquement, à la fin de l'effet.

Impressionnant, comme va en témoigner notre exemple. Copiez votre répertoire `highlight` vers un nouveau répertoire nommé `morph`, ajustez le HTML comme d'habitude et modifiez `applyEffect` comme suit :

Listing 8-10 Le script de démonstration pour `Effect.Morph`

```
function applyEffect(e) {
  e.stop();
  $('byebye').morph('font-size: 65px; color: #f08; opacity: 0.3;
    ↪ background: #ff8', { duration: 2 });
} // applyEffect
```

Un petit test dans votre navigateur vous montrera tout de suite ce que ça donne. Pour si peu de code, ça en jette, non ?

On peut encore monter d'un niveau avec la possibilité de définir de telles évolutions pour des groupes d'éléments, définis par sélection CSS ou par ID, et de déclencher ces transformations à volonté. C'est un besoin assez avancé, couvert par `Effect.Transform`.

SYNTAXE

```
new Effect.Transform([
  { selection1: style2 }[, ...]
][, options])
```

Nous allons voir un exemple en enrichissant un peu le précédent. Copiez votre répertoire `morph` vers un répertoire `transform`, ajustez le HTML comme d'habitude puis ajoutez-y la liste suivante :

Listing 8-11 Une petite liste pour améliorer la démonstration

```
<p id="byebye">Cette page teste l'effet Effect.Transform. Cliquez sur le
document pour avoir une démonstration.</p>

<ul class="morph">
  <li>À la une</li>
  <li>À la deux</li>
  <li>À la trois</li>
</ul>
```

Nous allons ajuster aussi le style :

Listing 8-12 Le style pour cette nouvelle liste

```
ul.morph {
  margin: 1em 0; padding: 0; list-style-type: none;
}
ul.morph li {
  text-align: center; color: green;
  margin: 0.25em 0; width: 10em; padding: 1em; background: #fcfc;
```

Enfin, nous ajustons le script comme ceci :

Listing 8-13 Le script de démonstration de Effect.Transform

```
var gTransformer;

function applyEffect(e) {
    e.stop();
    // Réinitialiser les styles individuels pour relancer la transfo.
    $$('#byebye, ul.morph li').each(function(e) {
        e.writeAttribute('style', '');
    });
    // Transformatioooooon !
    gTransformer.play();
} // applyEffect

function bindTestElements() {
    gTransformer = new Effect.Transform([
        { 'byebye': { paddingLeft: '50px', paddingRight: '50px',
            color: '#fff' } },
        { 'ul.morph li': { font-size: 20px; background: #fcc; color:
            '#800' } }
    ], { duration: 2 });
    document.observe('click', applyEffect);
} // bindTestElements
```

Remarquez que `Effect.Transform`, contrairement aux autres effets, ne se déclenche pas immédiatement : il faut appeler sa méthode `play()`, qu'on peut rappeler autant de fois qu'on le souhaite ! Cependant, comme après la première fois les éléments ont déjà leur valeur cible, nous réinitialisons dans `applyEffect` leur surcharge locale de style (réalisée par l'effet) avant de remettre le couvert...

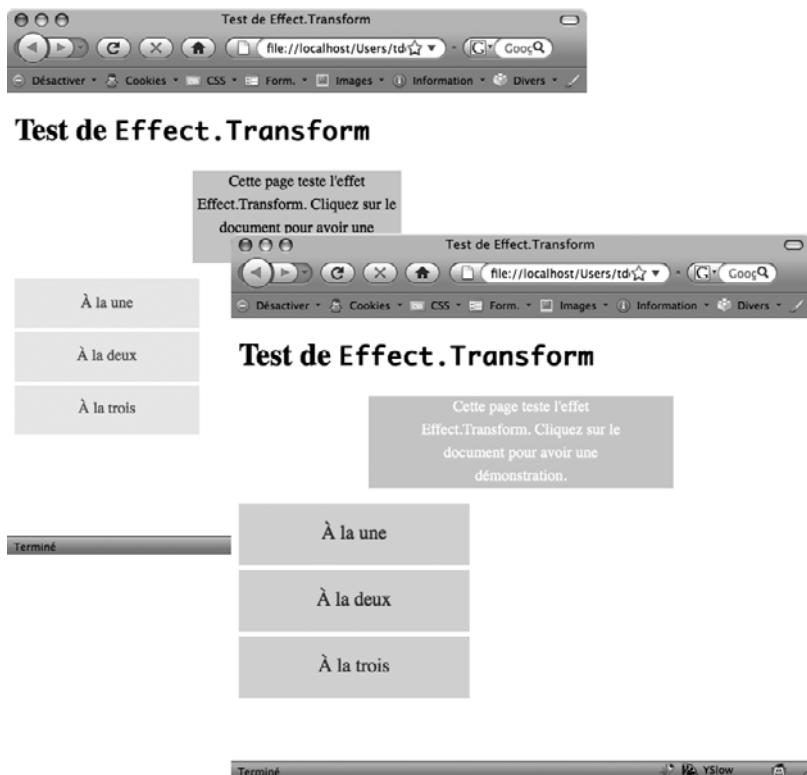
Voyez aussi la syntaxe de description des éléments ciblés et des styles à obtenir : un tableau d'associations uniques, avec la sélection comme clef et le style comme valeur. En interne, ceci utilise des `Effect.Morph`, donc toute valeur valide pour son option `style` est ici autorisée. Quant aux sélections, on tentera d'abord de s'en servir comme d'un ID ; si cela échoue, on les traitera comme une sélection CSS.

La figure 8-3 montre la page avant et après l'effet.

Je laisse de côté `Effect.Event` jusqu'à ce que nous étudions les files d'effets, car c'est là qu'il est utile...

Enfin, nous allons terminer en beauté avec un exemple un peu plus avancé : la combinaison d'effets synchronisés sur un même élément. On réalise ceci avec `Effect.Parallel`. De nombreux effets combinés (par exemple Puff, DropOut, Grow et Shrink) s'en servent.

Figure 8–3
Effect.Transform :
avant et après



Nous allons combiner un changement d'opacité et un rétrécissement, ce qui n'est pas sans rappeler `Effect.Puff`.

Copiez votre répertoire dans un nouveau répertoire parallel, ajustez le HTML et modifiez `applyEffect` comme suit.

Listing 8-14 Un exemple d'exécution en parallèle d'effets sur un même élément

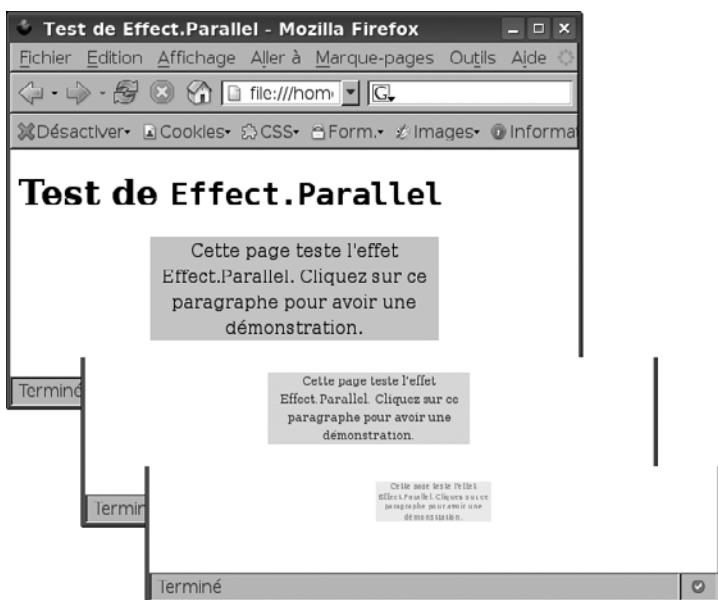
```
function applyEffect(e) {
    e.stop();
    new Effect.Parallel([
        new Effect.Opacity('byebye', { sync: true, from: 1, to: 0.33 }),
        new Effect.Scale('byebye', 40,
            { sync: true, scaleFromCenter: true })
    ], { duration: 2 });
} // applyEffect
```

Faites bien attention à la syntaxe ! Le constructeur prend deux arguments : un tableau d'effets et les options. Chaque effet doit par ailleurs activer son option `sync`,

comme indiqué plus haut. On a ici une réduction d'opacité jusqu'à 33 %, combinée à une réduction de taille jusqu'à 40 %.

L'exécution est impressionnante :

Figure 8-4
Notre effet combiné personnel !



À titre de bonus, voici un exemple encore plus avancé, très proche des effets combinés officiels, qui montre comment enchaîner un masquage de l'élément concerné par ces effets (en supposant qu'ils s'appliquent bien tous au même) une fois l'exécution parallèle terminée, à l'aide d'une fonction de rappel.

Listing 8-15 Ajout d'une fonction de rappel pour masquer finalement l'élément

```
function applyEffect(e) {
    e.stop();
    new Effect.Parallel([
        new Effect.Opacity('byebye', { sync: true, from: 1, to: 0.33 }),
        new Effect.Scale('byebye', 40,
            { sync: true, scaleFromCenter: true })
    ], {
        duration: 2,
        afterFinish: function(effect) {
            effect.effects[0].element.hide();
        }
    });
} // applyEffect
```

Souvenez-vous : les fonctions de rappel reçoivent l'effet conteneur (donc notre `Effect.Parallel`), lequel recense ses effets synchronisés dans une propriété tableau `effects`. Puisque nous savons ici que tous les effets synchronisés réfèrent au même élément, on prend le premier, et on appelle la méthode étendue `hide()` sur sa propriété `element`. Et voilà !

Les effets combinés

Les effets combinés sont constitués soit d'effets noyau spécialement paramétrés, soit d'exécutions parallèles de tels effets (réalisées avec `Effect.Parallel`), soit de séquences de tels effets (réalisées à l'aide de fonctions de rappel).

Attention, certains effets n'auront pas forcément le résultat désiré sur votre navigateur. La principale cause est la configuration d'une taille minimale pour les polices de caractères, qui va empêcher la réduction extrême de textes (ce qui affecte notamment `Shrink` et `Scale`). Soyez avertis !

À présent, dressons une liste rapide des 16 effets combinés. La plupart sont positionnels et fonctionneront donc mieux si vos éléments sont déjà positionnés, idéalement en absolu afin de les abstraire des contraintes de flux du contenu. Notez aussi qu'un effet combiné ne nécessite pas l'opérateur `new` pour son invocation.

Vous trouverez de nombreuses démonstrations légères de ces effets et des effets noyau dans les codes source de ma présentation à *The Ajax Experience West* en juillet 2007 : <http://tddsworld.com/confs/tae/script.aculo.us-demos.zip>.

Tableau 8–6 Les effets combinés de `script.aculo.us`

Effet	Description
<code>Effect.Appear</code>	Affiche l'élément en supprimant l'éventuel <code>display: none</code> de son attribut <code>style</code> , et en faisant graduellement passer son opacité de 0 % à 100 %. Hormis l'aspect <code>display</code> , équivalent à un <code>Opacity</code> . Voir <code>Fade</code> .
<code>Effect.BlindDown</code>	Affiche l'élément de haut en bas, sans modifier sa position dans la page ou sa présence. Le contenu de l'élément ne bouge pas, seule la zone de visualisation s'étend (contrairement à <code>SlideDown</code>).
<code>Effect.BlindUp</code>	Symétrique de <code>BlindDown</code> : masque l'élément en le rétrécissant de bas en haut.
<code>Effect.DropOut</code>	En plus d'un <code>Fade</code> , l'élément disparaît en « tombant ». Il part en quelque sorte aux oubliettes.
<code>Effect.Fade</code>	Symétrique de <code>Appear</code> : ramène l'opacité vers 0 % et ajoute un <code>display: none</code> au style en fin d'effet.
<code>Effect.Fold</code>	Fait disparaître un élément en réduisant d'abord sa hauteur, puis sa largeur, un peu comme on plierait (<i>fold</i>) une serviette en deux temps...
<code>Effect.Grow</code>	Amène l'élément d'une taille nulle à sa taille courante, en utilisant comme point d'origine une option <code>direction</code> (<code>top-left</code> , <code>top-right</code> , <code>bottom-left</code> , <code>bottom-right</code> , ou la valeur par défaut : <code>center</code>). Voir <code>Shrink</code> .

Tableau 8-6 Les effets combinés de script.aculo.us (suite)

Effet	Description
Effect.Puff	Donne l'illusion que l'élément part en fumée (mélange d'un Fade et d'un Grow en direction center) ! On préférera l'appliquer sur des éléments positionnés (en absolu ou relatif), pour éviter les décalages subis à gauche au déclenchement de l'effet.
Effect.Pulsate	Fait « pulser » l'élément, en un nombre défini de cycles Appear/Fade (sans finir par display: none, en revanche). Le nombre de cycles est défini par l'option pulses, qui vaut 5 par défaut.
Effect.ScrollTo	Anime un défilement du viewport (zone de la fenêtre qui affiche le document) pour que l'élément soit visible. Équivalent animé de la méthode scrollTo des éléments étendus, à ceci près que l'effet ne fait défiler que le <i>viewport</i> , et non le véritable conteneur de l'élément : à n'utiliser que pour les éléments qui ne sont pas dans un autre conteneur défilant, donc...
Effect.Shake	Déplace l'élément alternativement de droite à gauche, 6 fois, sur une durée par défaut d'une demi-seconde. L'amplitude du déplacement est gouvernée par l'option distance, qui vaut 20 (pixels) par défaut. Surtout utilisé pour signaler une saisie erronée dans un formulaire en « secouant » le champ concerné, voire le formulaire complet.
Effect.Shrink	Inverse de Grow : réduit l'élément jusqu'à disparition. Même option direction spécifique, mais qui indique ici un point de destination et non d'origine.
Effect.SlideDown	Similaire à BlindDown, si ce n'est que le contenu de l'élément suit le glissement, comme s'il était fixé sur un volet qu'on baissait. Attention : afin de pouvoir réaliser cet effet il faut un conteneur intermédiaire (typiquement un div ou un span) entre l'élément sur lequel on applique l'effet et son contenu.
Effect.SlideUp	Symétrique à SlideDown, mais fait disparaître le contenu de bas en haut. Même remarque que ci-dessus.
Effect.Squish	Écrase l'élément vers son coin supérieur gauche, et le fait finalement disparaître. Assez similaire à un Shrink avec direction: top-left.
Effect.SwitchOff	Simule l'extinction d'une ancienne télévision : un clignotement suivi d'un ramassement vers la ligne centrale. L'élément disparaît en fin d'effet.
Effect.toggle	Il ne s'agit pas tant d'un effet que d'une méthode de basculement d'état par effets (voir section suivante).

Effect.toggle

Cette fonction permet de faire basculer l'état d'un élément, de visible à invisible et inversement. Elle a un premier argument obligatoire, qui est bien sûr l'élément (ou son ID). Le deuxième argument est optionnel, et indique la famille d'effets à utiliser pour assurer la transition d'un état à l'autre. Trois valeurs sont possibles :

- 'appear', valeur par défaut, utilisera Appear et Fade.
- 'blind' utilisera BlindDown et BlindUp.
- 'slide' utilisera SlideDown et SlideUp.

De cette façon, nous n'avons pas à tester manuellement l'état affiché (propriété `display`) de l'élément dans nos scripts. Notez qu'il n'est pas nécessaire d'utiliser la même famille tout du long : un `Effect.toggle(element, 'blind')` affichera sans problème un élément préalablement caché par un `Effect.toggle(element)`, par exemple.

Quelques précisions importantes

Avant que vous ne vous jetiez sur vos tests personnels, permettez-moi de résumer les principaux points techniques à surveiller en manipulant ces effets :

- On ne le répétera jamais assez : pour masquer un élément d'entrée de jeu, il faut enfreindre légèrement la séparation du contenu et de la forme en définissant une propriété `display: none` dans son attribut `style`, et non dans une règle CSS. Sans quoi, `script.aculo.us` ne pourra pas l'afficher à nouveau.
- Pour les effets noyau, assurez-vous toujours d'utiliser `new` devant le nom de l'effet, afin d'éviter les surprises fâcheuses.
- La plupart des effets fonctionnent principalement sur des éléments de type bloc (voir l'annexe B pour plus de détails sur cette notion), à l'exception des éléments relatifs aux tables (par exemple `table`, `tr`, `td`, `thead`, `caption`), dont les valeurs pour la propriété `display` sont particulières et d'une gestion complexe, notamment sur MSIE.

Des commentaires qui font de l'effet

À titre de démonstration, nous allons reprendre notre dernier exemple de saisie de commentaires, que nous avons adapté au début de ce chapitre, pour y ajouter une petite dose d'effets qui ressembleront au comportement d'outils de blog récents.

Commencez par recopier votre répertoire `ajax_updater` dans un répertoire `fancy_comments`. Copiez ensuite dans le sous-répertoire `docroot` le fichier `effects.js` d'un des répertoires de test d'effet, par exemple `parallel`.

Commençons par ajuster notre HTML en ajoutant dans le `head` le chargement des effets, et en ajoutant un fragment pour l'affichage du nombre de commentaires.

Listing 8-16 Chargement de `script.aculo.us` et affichage du nombre de commentaires

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ➔ xml:lang="fr-FR">
```

```
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Des commentaires qui font de l'effet !</title>
  <link rel="stylesheet" type="text/css" href="client.css"> />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>

  ...

</form>

<p>Il y a <span id="commentCount">0 commentaire</span>.</p>

<div id="comments"></div>

</body>
</html>
```

Voici les modifications que nous souhaitons apporter :

- Nous affichons désormais le nombre de commentaires, comme vous avez pu le voir dans le listing ci-dessus. À chaque ajout, ce nombre clignotera, pour attirer l'attention sur la modification.
- Le nouveau commentaire apparaîtra parallèlement à l'aide d'un effet `BlindDown` (qui a l'avantage, par rapport à `SlideDown`, de ne rien exiger de particulier quant au XHTML de l'élément). En revanche, afin d'éviter un clignotement dû à son bref affichage suite à l'insertion dans le DOM, on va modifier notre modèle, `commentaire.rhtml`, pour utiliser un attribut `style` masquant d'entrée de jeu le bloc du commentaire.

Voici le nouveau modèle.

Listing 8-17 La version ajustée (extrait ciblé) du modèle de commentaire

```
<div class="comment" style="display: none;">
  ...
</div>
```

Et la nouvelle version de notre fonction switchToAjax !

Listing 8-18 L'ajout de commentaire, façon script.aculo.us

```
function switchToAjax(e) {
    ...
    new Ajax.Updater('comments', this.action,
    {
        parameters: this.serialize(),
        insertion: 'Bottom',
        onComplete: function() {
            var comments = $('comments').select('comment');
            var commentCount = comments.length;
            var lastComment = comments.last();
            var text = commentCount + ' commentaire';
            if (commentCount > 1)
                text += 's';
            $('commentCount').update(text);
            new Effect.Parallel([
                new Effect.Pulsate('commentCount', { sync: true }),
                new Effect.BlindDown(lastComment, { sync: true })
            ], { duration: 2 });
        }
    });
} // switchToAjax
```

Comme quoi, il y a des cas où onComplete est utile pour un Ajax.Updater.

En espérant que cela donne quelque chose une fois imprimé (et dans le cas contraire, qu'attendez-vous pour tester ?), la figure 7-4 vous donne un aperçu du résultat :

Files d'effets

Par défaut, lorsqu'on crée plusieurs effets, il s'exécutent en parallèle. Vous allez sûrement vous étonner, alors, qu'il existe un `Effect.Parallel`. Celui-ci est pourtant très utile, puisqu'il assure que les effets qu'il enrobe seront synchronisés : mêmes moments de démarrage et de fin d'exécution. Lancer plusieurs effets sans passer par lui les démarre un par un et les exécute à leur rythme : s'ils s'exécutent en parallèle, ils ne sont pas pour autant synchrones. Donc oui, c'est vrai, `Effect.Parallel` devrait plutôt se nommer `Effect.Synchronized`.

Figure 8–5
L'ajout de commentaire
avec effets



L'exécution simultanée est parfois problématique

Reste que cette exécution parallèle, qui est active par défaut, peut poser problème : certains effets vont entrer en conflit, et probablement aboutir à un aspect « cassé » du pauvre élément victime. Imaginez par exemple le code suivant :

```
new Effect.BlindUp('accountInfo');
new Effect.BlindDown('accountInfo');
```

Là où vous vouliez visiblement enchaîner les effets, leur exécution simultanée donne un résultat catastrophique (clignotement et, le plus souvent, élément invisible ou partiellement visible au bout du compte).

Les files, ou comment enchaîner les effets

Les files sont justement là pour permettre d'ordonner les effets. Dans `script.aculo.us`, on trouve un référentiel global de files nommé `Effect.Queues`, qui se comporte comme un tableau associatif d'objets `Effect.ScopedQueue`. Chaque file est nommée. Par défaut, il en existe une seule, nommée '`global`', qui est d'ailleurs accessible directement *via* la référence globale `Effect.Queue`. C'est dans cette file globale que vos effets sont ajoutés par défaut.

Quand un effet est ajouté à une file, il faut préciser notamment si on souhaite l'exécuter classiquement (sans planification particulière), ou le placer en début ou en fin de file. Ce détail est fourni par une option commune à tous les effets, que nous avons déjà évoquée : `queue`. Lorsque cette option manque, l'effet est ajouté pour exécution simultanée dans la file globale. Sinon, elle peut valoir '`front`', '`end`' ou '`withLast`' (les noms parlent d'eux-mêmes), ou être un objet anonyme précisant une file particulière (nous verrons cela dans un moment).

Ce qu'il faut bien comprendre, c'est le moment auquel un effet est exécuté : dès qu'une file a au moins un effet, elle va examiner régulièrement (dans la version actuelle, jusqu'à 60 fois par seconde, ce qui est largement suffisant) ses effets pour lancer ceux dont le moment de démarrage est arrivé, faire évoluer ceux dont le démarrage est passé, et retirer ceux qui auront terminé. Cette notion de moment de démarrage (propriété `startOn` de l'effet) est donc critique.

Par défaut, un effet démarre immédiatement (dès son premier examen par la file, au pire 15 ms après sa définition), ou après une attente de `delay` millisecondes, s'il est pourvu d'une option `delay`.

Mais l'utilisation d'une position explicite d'ajout dans la file modifie les moments de démarrage des effets de la file :

- pas d'information de position : ajout sans modifier quelque moment d'exécution que ce soit.
- '`front`' : ajout en début de file, et décalage des démarrages de tous les effets suivants (les effets qui avaient démarré très récemment risquent d'être suspendus !).
- '`end`' : ajout en fin de file, le moment de démarrage de l'effet étant ajusté pour pouvoir exécuter tous les autres avant.
- '`withLast`' : ajout en fin de file, le moment de démarrage étant synchronisé avec celui de l'effet qui était jusqu'à présent le dernier.

Il faut se méfier de files trop longues, par exemple en définissant une dizaine d'effets successivement pour la même file (ce qui serait probablement bien trop lourd visuellement en termes d'ergonomie, de toutes façons). En effet, il ne faut pas oublier qu'au bout de 15 ms maximum, la file va commencer à traiter ses effets. Ajouter un effet en position '`front`' plus de 15 ms après l'insertion du premier effet dans la file peut donner des

résultats inattendus... Si vous avez besoin de définir plus de 2 ou 3 effets dans une même séquence, prenez soin de les définir dans le bon ordre, en position 'end'.

Exemple d'enchaînement

Imaginons la séquence de définitions suivantes, en supposant que son exécution prendra moins de 15 ms :

```
new Effect.Highlight('userPad', { duration: 1.5 });
new Effect.Scale('userPad', { queue: 'end', delay : 2 });
new Effect.Appear('userPad', { queue: 'front' });
new Effect.Fade('notice', { queue: 'with-last' });
```

On obtient la file suivante :

- 1 Appear prévu pour exécution immédiate, durée par défaut 1s.
- 2 Highlight prévu pour exécution à 1 s (2 s après la fin de Appear).
- 3 Scale prévu pour exécution à 4,5 s (2 s après la fin de Highlight, de durée 1,5 s).
- 4 Fade prévu pour exécution à 4,5 s (synchrone avec le dernier jusqu'alors, soit Scale).

Utiliser plusieurs files

Ce n'est déjà pas mal, mais cela ne suffit pas toujours. En effet, on peut avoir plusieurs séquences d'effets, pour plusieurs portions de la page. Prenons un exemple, un peu forcé il est vrai : sur un site marchand, suite à l'ajout d'un achat dans le panier depuis une page de détails et au retour sur le catalogue, on pourrait avoir la séquence suivante :

- 1 Affichage et Highlight d'un message de confirmation en haut de page.
- 2 Trois secondes après la fin du Highlight, Fade de la confirmation.

Mais s'il s'agit du premier achat, on pourrait vouloir exécuter, parallèlement, la séquence suivante :

- 1 BlindDown du résumé de panier dans la barre latérale.
- 2 Une fois le BlindDown terminé, Appear du bouton de validation de commande sous le résumé.

Tout ceci a l'air attrayant, mais comment nous y prendre pour avoir deux files distinctes ? Rien de plus simple : il suffit d'en nommer au moins une. Pour pouvoir nommer la file à laquelle l'effet doit s'ajouter, il faut changer la nature de notre argument `queue` : au lieu d'une `String`, il va s'agir d'un objet anonyme avec une propriété `scope` (le nom de la file) et, si besoin est, une propriété `position` (toujours '`front`' ou '`end`').

Voici l'implémentation des besoins décrits ci-dessus.

Listing 8-19 Une définition avancée : deux files d'effets

```
// 1ère file
new Effect.Highlight('notice', { queue: { scope: 'notice' } });
new Effect.Fade('notice', {
    delay: 3, queue: { scope: 'notice', position: 'end' }
});

// 2ème file
new Effect.BlindDown('cart', { queue: { scope: 'cart' } });
new Effect.Appear('btnOrder', {
    queue: { scope: 'cart', position: 'end' }
});
```

Et voilà, ce n'était pas si compliqué !

Pour conclure, parlons de la surcharge de file. Lorsqu'un utilisateur sollicite trop l'interface (en cliquant à répétition sur un bouton, par exemple), on peut aboutir à un tel enchaînement d'effets qu'ils continuent inutilement leur exécution plusieurs secondes après que l'utilisateur... s'est calmé, dirons-nous. C'est souvent inutile.

Lorsque cela a du sens du point de vue de l'ergonomie, vous veillerez donc à poser une limite au nombre d'effets qui peuvent déjà se trouver dans la file avant d'insérer celui que vous êtes en train de définir. C'est tout simple : il suffit d'ajouter une propriété `limit` à la définition de file, par exemple :

```
new Effect.BlindDown('companyInfo', {
    queue: { scope: 'company', position: 'end', limit: 2 }
});
```

Glisser-déplacer

À présent que nous avons fait le tour des effets, il est temps de se pencher sur une autre fonctionnalité majeure de script.aculo.us : le glisser-déplacer. Ceux d'entre vous qui ont déjà tenté d'écrire leur propre gestion de glisser-déplacer en JavaScript portable ont goûté l'enfer. Avec script.aculo.us, plus de souci, une gestion solide est enfin disponible !

La gestion du glisser-déplacer dans script.aculo.us repose sur deux classes, que nous allons voir séparément : Draggable et Droppables. Ces classes sont fournies par le module `dragdrop.js`, qu'il faudra charger :

```
<script type="text/javascript" src="effects.js"></script>
<script type="text/javascript" src="dragdrop.js"></script>
```

Commençons par examiner les possibilités de glissement.

Faire glisser un élément avec Draggable

Pour pouvoir faire glisser un élément, il suffit de créer un objet `Draggable` basé sur l'élément, au chargement de la page. Le listing ci-après propose un exemple.

Listing 8-20 Altération d'un élément au chargement pour qu'il puisse glisser

```
function initDraggables() {
    new Draggable('myWizzyDiv');
} // initDraggables

document.observe('dom:loaded', initDraggables);
```

Comme vous pouvez le voir, c'est extrêmement simple. Précisons toutefois qu'on n'utilisera pas comme éléments des champs de formulaire, en tout cas pas directement (on utilisera par exemple leur paragraphe conteneur), en raison de problèmes sur certains navigateurs.

C'est donc simple, mais si la fonctionnalité s'arrêtait là, nous buterions rapidement sur ses limitations. On pourrait par exemple vouloir :

- limiter la zone au sein de laquelle l'élément peut être déplacé ;
- demander à l'élément de se déplacer par paliers d'un certain nombre de pixels (ce qu'on appelle un *snap*) plutôt que pixel par pixel ;
- demander à l'élément de revenir à sa position initiale une fois lâché ;
- altérer les effets visuels déclenchés à la prise et au relâchement ;
- modifier la position Z de l'élément pour qu'il soit masqué par certains autres (rarement utile, ceci dit) ;
- préférer limiter la zone de prise à un élément descendant donné plutôt qu'à toute la zone de l'élément concerné (pour réaliser une poignée, par exemple).

On le voit, les exigences concrètes peuvent être très variées sur de véritables projets. Heureusement pour nous, `Draggable` est à même de répondre à tous ces besoins (et même plus) ! Il utilise pour cela des options, passées de façon classique, comme propriétés d'un objet anonyme fourni en deuxième argument. Les propriétés disponibles sont les suivantes.

Tableau 8–7 Propriétés prises en charge par Draggable

Option	Description
<code>zindex</code>	Position Z de l'élément (1 000 par défaut, ce qui assure a priori que l'élément est toujours visible).
<code>revert</code>	Demande à l'élément d'exécuter <code>reverteffect</code> une fois relâché (<code>false</code> par défaut). On peut aussi simplifier en précisant directement la fonction à invoquer, et laisser <code>reverteffect</code> tranquille. Si l'on gère des zones de dépôt, on pourra aussi préciser ici ' <code>failure</code> ', mais nous y reviendrons.
<code>snap</code>	Gère l'ajustement de la position au fil du glissement. Peut prendre de nombreuses formes : – <code>false</code> (valeur par défaut) : pas d'ajustement ou de limitation. – nombre entier : taille du <code>snap</code> en pixels, par exemple 10 pour se déplacer par paliers de 10 pixels. – Tableau de deux nombres entiers : différenciation horizontale et verticale, par exemple [10, 20] utilisera un palier horizontal de 10 et un palier vertical de 20. – Fonction : prend la position prévue en arguments (X, Y) et renvoie un tableau avec la position ajustée [ax, ay]. Permet d'assurer un <code>snap</code> uniquement à certains endroits, par exemple au bord de certains éléments, et de limiter la zone de glissement autorisée !
<code>handle</code>	Permet de définir un élément servant de poignée (<code>handle</code>) pour le glissement. Par défaut vaut <code>null</code> , de sorte que toute la surface de l'élément est utilisable pour démarrer le glissement. Peut prendre plusieurs autres valeurs : – nom de classe CSS : le premier élément fils ayant cette classe servira de poignée ^a ; – ID d'élément ou élément directement : désigne l'élément devant servir de poignée. Une utilisation classique consiste à limiter le déclenchement du glissement à une «barre de titre» pour un élément, généralement représentée sous forme d'un élément <code>div</code> ou de titre (<code>hx</code>) fils.
<code>constraint</code>	Peut valoir 'vertical' ou 'horizontal', ce qui limite la direction autorisée pour le glissement. Indéfini par défaut.
<code>ghosting</code>	Si actif, déplace un clone de l'objet plutôt que l'objet lui-même, jusqu'au dépôt. Vaut <code>false</code> par défaut.
<code>starteffect</code>	Fonction appelée au démarrage du glissement. Par défaut, sauvegarde l'opacité actuelle de l'élément puis l'amène à 70 % à l'aide d'un <code>Effect.Opacity</code> . Reçoit l'élément en argument.
<code>endeffect</code>	Fonction appelée au relâchement. Par défaut, restaure l'opacité sauvegardée (100 % si elle n'a pas été déterminée au démarrage) à l'aide d'un <code>Effect.Opacity</code> . Reçoit l'élément en argument. En la définissant explicitement à <code>false</code> , on désactive les effets de début et de fin par défaut.
<code>reverteffect</code>	Fonction appelée au relâchement, après <code>endeffect</code> , si la propriété <code>revert</code> est à <code>true</code> . Par défaut, ramène l'élément à sa position d'origine, en un temps proportionnel à la distance de glissement, à l'aide d'un <code>Effect.Move</code> . Reçoit trois arguments : l'élément et les composantes verticale et horizontale du glissement.
<code>scroll</code>	Indique si le glissement doit, lorsqu'il atteint la limite d'affichage d'un élément conteneur capable de défiler, déclencher le défilement de cet affichage pour pouvoir continuer à glisser. Vaut <code>false</code> par défaut, mais peut référencer un objet conteneur par son ID ou

Tableau 8-7 Propriétés prises en charge par Draggable (suite)

Option	Description
	directement, par exemple l'objet global window. Du coup, en atteignant le bord de la page, celle-ci va se mettre à défiler (même si elle n'en avait pas besoin, car tout son contenu était visible !) pour permettre de prolonger le glissement.
scrollSensitivity	Niveau de proximité aux bords du conteneur pour déclenchement du défilement. En pixels ; vaut 20 par défaut.
scrollSpeed	Vitesse de défilement, en pixels par progression du glissement. Vaut 15 par défaut.
delay	Temps en millisecondes pendant lequel le bouton de la souris doit être enfoncé avant que le glisser-déplacer ne puisse avoir lieu. Vaut zéro (désactivé) par défaut. Cette option et son équivalent sur Sortable.create sont les seules en millisecondes de tout Prototype et script.aculo.us !

a. Cette possibilité semble défectueuse dans la version 1.6.2...

Eh bien, que d'options ! Réalisons un petit exemple pour nous y retrouver.

Commencez par copier votre répertoire opacity dans un nouveau répertoire draggable. Ajoutez-y le fichier dragdrop.js de la bibliothèque. Nous allons ensuite modifier la page HTML pour préparer notre exemple.

Listing 8-21 Notre page HTML avec un pavé déplaçable

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Test de Draggable</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="dragdrop.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test de <code>Draggable</code></h1>

<div id="wizzy">
  <div id="wizzy-menubar"></div>
  <div class="contents">Faites-moi glisser !</div>
</div>
```

```
</body>  
</html>
```

La feuille de styles, `tests.css`, est assez simple.

Listing 8-22 La feuille de styles pour notre exemple

```
#wizzy {  
    width: 25ex;  
    border: 1px solid navy;  
    background: #ddf;  
    color: blue;  
    text-align: center;  
    font-size: large;  
}  
  
.contents {  
    padding: 1em;  
}
```

Nous obtenons l'aspect suivant.

Figure 8-6

Notre page avant qu'on fasse glisser le bloc



Notez la partie haute du pavé, sa « barre de titre » en quelque sorte, qui va servir de seule zone possible pour déclencher le glissement, comme on le voit dans le script suivant, `tests.js` :

Listing 8-23 Notre script de tests

```
MAX = 400;

function initDraggables() {
    new Draggable('wizzy', {
        revert: true, handle: 'wizzy-menubar',
        snap: function(x, y) {
            return [ [x, MAX].min(), [y, MAX].min() ];
        }
    });
} // initDraggables

document.observe('dom:loaded', initDraggables);
```

Pour cet exemple, nous utilisons :

- revert, pour demander à l'élément de revenir à sa position initiale une fois relâché.
- handle, pour préciser l'identifiant de l'élément qui servira de zone de déclenchement. Cliquer sur le corps du pavé ne servira donc à rien, comme c'est habituellement le cas pour la plupart des systèmes de fenêtrage.
- snap, d'une façon avancée : nous définissons notre propre fonction de contrôle, qui limitera les déplacements à la position (400, 400), donc des limites droite et basse.

Le pavé bénéficie des comportements par défaut pour les effets de démarrage et d'arrêt (ajustement de l'opacité à 70 %), ainsi que pour le relâchement (retour à la position de départ).

Chargez la page et amusez-vous à déplacer le bloc !

Figure 8-7

Notre pavé en cours de déplacement



Désactiver la possibilité de faire glisser un élément

Si vous souhaitez empêcher, temporairement ou définitivement, un élément de glisser, il suffit de conserver une référence sur le Draggable que vous créez pour lui, et de la passer par la suite à Draggables.unregister (attention au « s » : c'est un pluriel). Vous pourrez réactiver le glissement plus tard en appelant Draggables.register (ce qui avait été fait automatiquement quand vous aviez créé le Draggable).

Réagir aux étapes des glissements

Il est possible d'inscrire des observateurs auprès de Draggables, à l'aide de sa méthode addObserver (et bien sûr, on peut se désinscrire avec removeObserver). Chaque observateur peut implémenter jusqu'à trois méthodes correspondant aux trois catégories d'événement dans un glissement : onStart, onDrag et onEnd. Seule onDrag peut être appelée plusieurs fois (chaque fois que le glissement évolue dans l'espace).

Les trois méthodes acceptent trois arguments :

- 1 Le nom de l'événement ('onStart', 'onDrag' ou 'onEnd'), ce qui permet par exemple d'utiliser une même méthode pour plusieurs événements.
- 2 L'objet Draggable concerné (puisque'on enregistre les observateurs au niveau global, avec Draggables.addObserver).
- 3 L'objet événement, manipulable avec le Event de Prototype.

Voici un exemple simple, utilisant l'objet global console de Firebug :

```
var observer = {
    onStart: function(eventName, draggable) {
        console.log(draggable.id + ' : ' + eventName);
    }
};
observer.onEnd = observer.onStart;
Draggables.addObserver(observer);
```

Gérer le dépôt d'un élément avec Droppables

Faire glisser, c'est bien, mais si c'est pour ne pas déposer dans un endroit précis, ça ne sert qu'à réarranger la page (ce qui n'est déjà pas si mal).

Il est bien entendu possible, avec script.aculo.us, de définir des éléments de la page comme étant des zones de dépôt (ce que la bibliothèque appelle des *droppables*).

Chaque zone peut préciser certaines conditions d'acceptation pour un dépôt (par exemple, « uniquement les éléments ayant la classe X » ou « uniquement ceux issus du conteneur Y »), réagir au survol d'un candidat valide en cours de déplacement, et bien sûr, réagir au dépôt à proprement parler.

Il ne s'agit pas ici de créer un objet d'enrobage, mais simplement d'inscrire ou de désinscrire notre élément dans le référentiel global. Là où on avait Draggable, Draggables.register et Draggables.unregister, on a ici juste Droppables.add et Droppables.remove.

L'ajout prend en charge plusieurs options. Toutes, sauf onDrop, sont optionnelles.

Tableau 8–8 Options prises en charge pour l'inscription d'une zone de dépôt

Option	Description
accept	Nom ou tableau de noms de classes CSS, qui établit un filtre : seuls les éléments disposant d'au moins une des classes spécifiées auront le droit d'être déposés.
containment	Référence ou tableau de références d'éléments (ID ou références directes), qui établit un filtre lui aussi : seuls les éléments contenus dans un des conteneurs spécifiés auront le droit d'être déposés. Cumulatif avec accept.
hoverClass	Classe CSS ajoutée à la zone de dépôt lorsqu'un élément acceptable est en train de glisser dessus. Pratique pour indiquer que le dépôt est autorisé.
onDrop	Seule option obligatoire, doit fournir une fonction de gestion du dépôt, qui va probablement synchroniser la couche serveur via Ajax et récupérer un ajustement de la page en réponse.

Il existe en réalité quelques options supplémentaires, mais qui sont principalement utilisées en interne par Sortable, le mécanisme de tri dynamique d'éléments par glisser-déplacer que nous verrons à la prochaine section.

Un exemple sympathique de glisser-déposer

À titre d'exemple, nous allons réaliser un équivalent de la démonstration de panier présente sur le site de script.aculo.us, avec une couche serveur qui composera à chaque manipulation le contenu de la zone « panier » (de là à ajouter l'information dans une gestion de sessions, il n'y aurait qu'un pas).

Le principe est simple : nous avons quelques éléments à vendre dans notre magasin, à savoir des tasses et des tee-shirts (le parfait magasin en ligne pour geeks). Pour placer un élément dans le panier, nous allons simplement le faire glisser. Ajouter plusieurs fois le même produit fera l'objet d'une gestion de quantité par type de produit. Pour retirer un produit, il suffira de le faire glisser vers une poubelle à côté du panier.

Copiez donc votre répertoire draggable dans un nouveau répertoire fancy_cart/docroot. Vous pourrez trouver les images de cet exemple dans l'archive des codes source disponible sur le site des éditions Eyrolles, et les déposer dans le répertoire. Commençons par modifier notre HTML.

Listing 8-24 La page HTML pour notre gestion de panier

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  > xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    > charset=iso-8859-15" />
  <title>Un sympathique panier</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="dragdrop.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Un sympathique panier</h1>

<div id="products">
  
  
</div>

<h2>Votre panier </h2>

<div id="cart" class="cart">
  <div id="items">Votre panier est vide.</div>
  <div id="cartFooter">
    <p id="wastebin"></p>
    <p id="indicator" style="display: none">
      
      Mise à jour…;
    </p>
  </div>
</div>

</body>
</html>
```

La feuille de styles est plus longue que d'habitude mais reste assez simple (et comme toujours, dans le doute, commencez par l'annexe B).

Listing 8-25 La feuille de styles pour notre gestion de panier

```
body {  
    background: white;  
}  
  
#products {  
    margin-bottom: 20px;  
    height: 120px;  
}  
  
#cart {  
    margin-top: 10px;  
}  
  
#cartFooter {  
    position: relative;  
}  
  
#indicator {  
    position: absolute;  
    left: 90px;  
    top: 0;  
    height: 16px;  
    font-family: sans-serif;  
    font-size: small;  
    color: gray;  
}  
  
#indicator img {  
    vertical-align: middle;  
}  
  
#items {  
    width: 500px;  
    height: 96px;  
    border: 1px solid orange;  
    padding: 1ex;  
    font-family: sans-serif;  
    color: gray;  
}  
  
#items.active {  
    background: #fec;  
}
```

```
#wastebin {  
    margin: 0;  
    width: 64px;  
    padding: 1ex;  
    border: 2px dotted white; /* MSIE ignore 'transparent' */  
}  
  
.cartItemsLine {  
    font-size: small;  
}  
  
.cartItemsLine img {  
    height: 32px;  
    vertical-align: middle;  
}  
  
.product {  
    cursor: move;  
}
```

Voici à présent l'un des deux gros morceaux de l'exemple : le script côté client.

Listing 8-26 Notre script client pour ce sympathique panier

```
function initDraggables() { ❶  
    new Draggable('product_1', { revert: true });  
    new Draggable('product_2', { revert: true });  
    Droppables.add('items', { ❷  
        accept: 'product', hoverclass: 'active',  
        onDrop: updateCart.curry('add')  
    });  
    Droppables.add('wastebin', {  
        accept: 'cartItem', hoverclass: 'active',  
        onDrop: function(product) {  
            product.hide();  
            updateCart('remove', product);  
        }  
    });  
} // initDraggables  
  
function toggleIndicator() { ❸  
    $('#indicator').toggle();  
} // toggleIndicator
```

```
function updateCart(mode, product) { ④
    var id = product.id.split('_')[1];
    new Ajax.Updater('items', '/' + mode, {
        postBody: { 'id': id },
        evalScripts: true,
        onLoading: toggleIndicator,
        onComplete: toggleIndicator
    });
} // updateCart

document.observe('dom:loaded', initDraggables);
```

- ➊ On commence par déclarer que nos deux produits peuvent être déplacés, et doivent se repositionner une fois lâchés.
- ➋ On inscrit l'afficheur du panier comme zone de dépôt réservée aux produits, et la poubelle comme zone de dépôt réservée aux éléments du panier (vous verrez que le XHTML renvoyé par le serveur accole une classe `cartItem` aux éléments qui peuvent être déplacés). Pour le coup, il était utile de factoriser l'invocation Ajax...
- ➌ Simple bascule d'affichage pour notre indicateur (rappel : son `display: none` doit être *inline*, pas dans la CSS !).
- ➍ L'invocation Ajax, assez avancée ! On appelle soit `/add` soit `/remove`, avec toujours un argument `id` qui identifie le produit, le tout en `post` par défaut. L'appel est encadré par les bascules de visibilité de l'indicateur. Comme on va renvoyer du XHTML contenant du JavaScript, il ne faut pas oublier d'activer `evalScripts` !

Remarquez au passage que, pour tant de traitements, ce script est vraiment court ! Prototype et script.aculo.us sont de bonnes bibliothèques...

Et bien sûr, il reste à écrire la gestion côté serveur. Nous déléguerons la création du fragment XHTML représentant l'intérieur du panier à un modèle interprété par ERb, comme nous l'avons déjà fait plusieurs fois. Voici d'abord le code serveur, dans un fichier `serveur.rb` au-dessus de `docroot`.

Listing 8-27 Le script serveur de gestion du panier, `serveur.rb`

```
#!/usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

PRODUCT_LABELS = { ①
    '1' => 'Mug',
    '2' => 'T-shirt'
}
```

```

cart = {}

template_text = File.read('cart.rhtml')
cart_html = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/add') do |request, response| ②
  product_id = CGI::parse(request.body)['id'][0]
  cart[product_id] = cart.include?(product_id) ? cart[product_id] + 1 : 1
  sleep 1 # Simuler un A/R web, qu'on voie l'indicateur...
  response['Content-Type'] = 'text/html'
  response.body = cart_html.result(binding)
end

server.mount_proc('/remove') do |request, response| ③
  product_id = CGI::parse(request.body)['id'][0]
  if cart[product_id] > 1
    cart[product_id] = cart[product_id] - 1
  else
    cart.delete(product_id)
  end
  sleep 1
  response['Content-Type'] = 'text/html'
  response.body = cart_html.result(binding)
end

trap('INT') { server.shutdown }

server.start

```

- ① Évidemment, dans une véritable application, on irait pêcher ces noms dynamiquement dans la base de données en réponse aux actions add et remove... Notez également l'initialisation d'un unique panier indépendant du client, là où on utiliserait des cookies de session en temps normal.
- ② Ajout de produit : on récupère le paramètre id transmis en post, puis si on a déjà cet ID dans le panier, on augmente la quantité, sinon on le crée avec une quantité de un.
- ③ Suppression de produit : on récupère l'ID, puis s'il y en a plus d'un dans le panier, on diminue la quantité, sinon on retire carrément le produit du panier.

Et voici le modèle, `cart.rhtml`, dans le même répertoire. Il est un peu plus complexe que d'habitude, aussi nous allons détailler.

Listing 8-28 Le modèle utilisé pour construire le XHTML interne du panier

```

<% cart.each do |product, qty| %> ①
  <div class="cartItemsLine">
    <% qty.times do |i| %> ②
      _<%= i %>" style="position: relative" />
      <script type="text/javascript">
        new Draggable('item_<%= product %>_<%= i %>', { revert: true });
      </script>
    <% end %> ④
    <span class="title">
      <%= PRODUCT_LABELS[product] + " (#{$qty})" %>
    </span>
  </div>
<% end %> ⑤
<%= 'Votre panier est vide.' if cart.empty? %> ⑥

```

- ① Boucle sur les produits du panier, avec leur ID et quantité.
- ② Boucle de 0 à quantité – 1.
- ③ On crée autant d'images du produit que nécessaire. Notez la classe `cartItem` et l'ID unique, utilisé dans le script pour pouvoir déplacer l'image.
- ④ Fin de la boucle sur quantité.
- ⑤ Fin de la boucle sur les produits.
- ⑥ N'oublions pas les paniers vides !

Ouf ! Voilà un exemple massif ! Pour pouvoir le tester, lancez le serveur avec `ruby serveur.rb`, et naviguez sur `http://localhost:8042/`

Précautions d'emploi

Encore tous joyeux d'avoir réalisé un aussi joli exercice, vous imaginez déjà les interfaces riches que vous allez réaliser dans les tous prochains jours... Prêtez tout de même attention à ces quelques remarques importantes :

- Si vous deviez retirer une zone de dépôt du DOM, il est impératif que vous le signaliez au préalable au référentiel `Droppables`, à l'aide d'un appel à `Droppables.remove`. Si vous oubliez cette précaution, toute tentative de glisser-déplacer ultérieure échouera.
- Si vous avez une interface suffisamment (inutilement ?) complexe pour avoir des zones de dépôt imbriquées, vous devez absolument les enregistrer de l'intérieur vers l'extérieur, sous peine de ne voir réagir que les zones externes, aux dépends des zones imbriquées.

Figure 8–8

Notre panier, vierge,
au démarrage

**Figure 8–9**

Déplacement d'un produit dans
le panier pour commande



Figure 8–10

Le panier mis à jour

**Figure 8–11**

Le panier après quelques ajouts supplémentaires. Notez les quantités.



Figure 8-12

Retrait d'un produit du panier



Le dépôt, côté Draggable

Lorsqu'on a mis en place des zones de dépôt, les objets Draggable concernés disposent d'une valeur d'option et d'une fonction de rappel complémentaires :

- revert peut avoir la valeur spéciale 'failure'. Cela signifie que le revertEffect ne sera déclenché qu'en cas de dépôt hors d'une zone de dépôt valide.
- La fonction de rappel spécifique onDropped est appelée avec l'élément déplacé en argument. Elle survient après le onDrop de la zone de dépôt, mais avant le onEnd du Draggable.

Tri de listes par glisser-déplacer

Inutile de nous arrêter là, le glisser-déplacer a une utilité de premier plan dans une interface web : l'ordonnancement d'éléments dans une liste ! Sans glisser-déplacer, on en est réduit à cliquer à qui mieux mieux sur des boutons de montée et de descente en face de chaque élément. Lorsqu'on n'a vraiment pas de chance, chaque clic recharge la page. C'est comme ça qu'on obtient des listes non triées, la flemme ayant eu raison des bonnes intentions de l'utilisateur.

Mais avec le glisser-déplacer, tout va mieux, l'utilisateur retrouvant le confort habituel de ses applications classiques. C'est une utilisation tellement courante du glisser-déplacer que script.aculo.us fournit, évidemment, un mécanisme prêt à l'emploi : `Sortable`. Le mode d'emploi général est plutôt simple : on indique qu'il est possible de trier une structure avec un appel à `Sortable.create`, on désactive la fonction en appelant `Sortable.destroy`, et en cadeau bonus (parce que vous êtes sympathique), on obtient une représentation toute prête à l'envoi vers le serveur en appelant `Sortable.serialize` ! Voyons ces fonctions dans le détail.

Que peut-on trier ?

On peut théoriquement trier le contenu de n'importe quel élément de type bloc, à l'exception des conteneurs de table, comme d'habitude (plus spécifiquement les éléments `table`, `thead`, `tbody`, `tfoot` et `tr`). Il semble toutefois qu'en utilisant un `tbody` signifié comme conteneur, les éléments `tr` à l'intérieur puissent être ordonnés correctement sur MSIE et Firefox.

Autre contrainte liée aux tables : un conteneur dans lequel on peut trier posera problème sur MSIE s'il est présent dans une table, à moins que la table ait une propriété CSS `position: relative`.

On peut donc trier les éléments fils d'une liste bien sûr (`ol`, `ul`) mais aussi de n'importe quel conteneur, par exemple des `p` dans un `div`. On peut même jouer sur l'horizontalité aussi, avec des éléments fils de type en ligne : `img`, `span`, etc. ou des éléments flottants.

Activer les fonctions d'ordonnancement

Pour pouvoir manipuler le contenu d'un élément en vue d'un ordonnancement, il faut appeler `Sortable.create` sur cet élément conteneur. C'est sans doute l'appel script.aculo.us qui a le plus d'options jusqu'ici, à l'exception des effets noyau. Jugez plutôt !

Tableau 8–9 Options prises en charge par `Sortable.create`

Option	Description
<code>tag</code>	Nom des balises pour les éléments fils qui vont pouvoir être déplacés. Par défaut <code>li</code> , ce qui évite de le préciser dans le cas fréquent des listes. Pour les autres cas, à vous de le préciser.
<code>only</code>	Filtre supplémentaire optionnel pour les éléments fils qu'on peut déplacer, d'une syntaxe identique à l'option <code>accept</code> de <code>Droppables.add</code> : nom de classe CSS ou tableau de noms de classes.
<code>elements</code>	Version optimisée court-circuitant <code>tag</code> et <code>only</code> , et listant directement les éléments pouvant être déplacés au sein du conteneur. Gros gain de performance sur de grandes quantités d'éléments. Voir aussi <code>handles</code> .

Tableau 8–9 Options prises en charge par Sortable.create (suite)

Option	Description
overlap	Indique la direction de l'ordonnancement. Par défaut 'vertical', mais pour des éléments flottants ou des listes horizontales, préciser 'horizontal'.
constraint	Identique à l'option homonyme de Draggable, mais ici à 'vertical' par défaut. Pour retirer toute contrainte, on remettra donc à <code>false</code> .
containment	Détermine le ou les conteneurs au sein desquels les éléments fils peuvent être déplacés. Par défaut, concerne uniquement le contenu sur lequel on appelle Sortable.create. Mais on peut passer un tableau de conteneurs ou d'ID de conteneurs (qui doit obligatoirement inclure le conteneur courant), par exemple pour permettre l'échange entre plusieurs listes.
handle	Identique à l'option homonyme de Draggable.
handles	Version optimisée consistant à fournir directement un tableau des <code>handles</code> pour les éléments à ordonner. Gros gain de performance sur de grandes quantités d'éléments. Voir aussi <code>elements</code> .
hoverClass	Identique à l'option homonyme de Droppables.add.
ghosting	Identique à l'option homonyme de Draggable.
dropOnEmpty	Si actif, rend le conteneur Droppable lorsqu'il devient vide, en respectant <code>containment</code> pour les conteneurs sources. N'a pas d'intérêt si <code>containment</code> n'a que le conteneur courant, donc par défaut à <code>false</code> .
scroll	Analogue à l'option homonyme de Draggable, mais limité au conteneur courant, qui aura de préférence la propriété <code>overflow: scroll</code> .
scrollSensitivity	Identique à l'option homonyme de Draggable.
scrollSpeed	Identique à l'option homonyme de Draggable.
onChange	Fonction de rappel invoquée dès que l'ordre des éléments change. Reçoit l'élément déplacé en argument. Lors d'un transfert entre deux conteneurs, elle est appelée sur les deux.
onUpdate	Fonction de rappel invoquée en fin de glissement d'un élément, si l'ordre résultant a effectivement changé. Reçoit le conteneur en argument. Lors d'un transfert entre deux conteneurs, elle est appelée sur les deux. Les éléments fils doivent avoir des ID nommés comme l'exige Sortable.serialize (voir plus loin).
delay	Temps en millisecondes pendant lequel le bouton de la souris doit être enfoncé avant que le glisser-déplacer ne puisse avoir lieu. Vaut zéro (désactivé) par défaut.

Il existe par ailleurs deux options depuis la version 1.6.1, `tree` et `treeTag`, qui permettent la gestion d'arborescences plutôt que de listes à plat. Elles sont assez expérimentales et n'ont jamais été totalement testées. Pour des exemples d'utilisation, voyez les tests fonctionnels correspondants dans les sources de la bibliothèque.

Réalisons un premier exemple avec une seule liste. Copiez votre répertoire `draggable` dans un nouveau répertoire `sortable_one_list`. Nous allons modifier le HTML comme suit.

Listing 8-29 La page de test pour notre premier exemple d'ordonnancement

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Test de Sortable (une liste)</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="dragdrop.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test de <code>Sortable</code> (une liste)</h1>

<ul id="people">
  <li id="person_1">Acyा</li>
  <li id="person_2">Adrien</li>
  <li id="person_3">Annabelle</li>
  <li id="person_4">Christophe</li>
  <li id="person_5">Élodie</li>
  <li id="person_6">Guillaume</li>
</ul>

</body>
</html>
```

Un peu de CSS pour améliorer la présentation de la liste...

Listing 8-30 Quelques règles CSS font des merveilles sur notre liste

```
#people {
  padding: 0;
}

#people li {
  font-family: sans-serif;
  list-style-type: none;
  width: 20ex;
  height: 1.5em;
  line-height: 1.5em;
  padding: 0.5ex;
  margin: 0.5ex 0;
```

```

background: #ffa;
border: 1px solid #880;
cursor: move;
}

```

Voyons à présent le script, outrageusement simple.

Listing 8-31 Notre script activant l'ordonnancement pour une liste

```

function initSortables() {
    Sortable.create('people');
} // initSortables

document.observe('dom:loaded', initSortables);

```

En dépit (ou plutôt à cause) de sa simplicité, ce code soulève une question ardue : est-il vraiment moral de faire payer le client pour ça ? Mais baste, laissons de côté ces considérations éthiques, et observons le résultat.

Figure 8-13

Déplacement dans une liste où le tri est possible



C'est sympathique, non ? Voyons un peu la différence avec le *ghosting* activé. Modifiez simplement les options de *create* :

```

Sortable.create('people', { ghosting: true });

```

Rechargez et tentez un déplacement.

Figure 8-14

Déplacement « ghost » dans une liste où le tri est possible



Vous voyez comme l'élément reste à sa place en attendant le relâchement, laissant un clone translucide servir au déplacement ? Je trouve ça plus agréable quand on transfère d'une liste à l'autre. Et justement...

Réalisons un deuxième exemple, avec deux listes aux éléments interchangeables ! Copiez votre répertoire dans un nouveau répertoire `sortable_two_lists`. On commence, comme d'habitude, par modifier le HTML.

Listing 8-32 Notre page HTML avec deux listes

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  >>> xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    >>> charset=iso-8859-15" />
  <title>Test de Sortable (deux listes)</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="dragdrop.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>
<body>
```

```
<h1>Test de <code>Sortable</code> (deux listes)</h1>

<ul id="side1" class="people">
    <li id="person_1">Acyा</li>
    <li id="person_2">Annabelle</li>
    <li id="person_3">Christophe</li>
    <li id="person_4">Élodie</li>
    <li id="person_5">Guillaume</li>
</ul>

<ul id="side2" class="people">
    <li id="person_11">Amir</li>
    <li id="person_12">Aurore</li>
    <li id="person_13">Claude</li>
    <li id="person_14">Jacques</li>
    <li id="person_15">Janusz</li>
    <li id="person_16">Julien</li>
</ul>

</body>
</html>
```

On va ajuster un tout petit peu la CSS pour traiter plusieurs listes similaires.

Listing 8-33 Notre CSS à jour pour traiter deux listes

```
.people {
    position: absolute;
    padding: 0;
}
.people li {
    font-family: sans-serif;
    list-style-type: none;
    width: 20ex;
    height: 1.5em;
    line-height: 1.5em;
    padding: 0.5ex;
    margin: 0.5ex 0;
    border: 1px solid #880;
    cursor: move;
}
#side1 li {
    background: #ffa;
}
#side2 {
    left: 30ex;
}
```

```
#side2 li {  
    background: #cfa;  
}
```

Et enfin, on ajuste notre script, ce qui permet de voir quelques options supplémentaires (au passage, on factorise pour la première fois les options).

Listing 8-34 Notre script pour deux listes aux éléments interchangeables

```
function initSortables() {  
    var options = {  
        dropOnEmpty: true, containment: [ 'side1', 'side2' ],  
        constraint: false, ghosting: true  
    };  
    Sortable.create('side1', options);  
    Sortable.create('side2', options);  
} // initSortables  
  
document.observe('dom:loaded', initSortables);
```

Voyons un peu le résultat...

Figure 8-15

Déplacement interliste,
avec « ghosting » !



Remarquez que les éléments de liste prennent évidemment la couleur de leur liste conteneur, en vertu des règles CSS applicables.

Figure 8–16
Nos listes après
quelques transferts...



Il nous reste toutefois un petit problème : nous avons beau avoir activé l'option `dropOnEmpty`, elle ne va pas nous servir à grand-chose. En effet, une fois une des listes totalement transférée dans l'autre, les dimensions calculées par le navigateur pour celle-ci se réduisent à zéro. Il n'y a donc plus rien sur quoi ramener des éléments !

Voici une proposition de solution : nous allons définir une classe supplémentaire applicable aux listes, qui leur donne une couleur de fond et de bordure identifiables, et nous allons définir des dimensions par défaut pour les listes, qui seront certes dépassées (notamment en hauteur) l'essentiel du temps (car la valeur par défaut pour la propriété `overflow` est `visible`), mais nous seront bien utiles une fois l'une des listes vide. Voici déjà les ajustements à la CSS, mis en exergue.

Listing 8–35 Ajustements à notre CSS pour gérer les listes vides

```
.people {
    position: absolute;
    padding: 0;
    width: 22ex;
    height: 2em;
}
.people.empty {
    background: #eee;
    border: 1px solid silver;
}
...
```

Mais cela ne suffit pas : certes, une fois la liste vide, elle occupera toujours un certain espace, et on pourra déposer dessus, mais faute de couleurs pour la rendre évidente, l'utilisateur ne saura pas qu'il peut y déposer quoi que ce soit. Or, la classe à appliquer doit l'être dynamiquement. C'est l'occasion rêvée de s'essayer à la fonction de rappel onUpdate.

Listing 8-36 Notre nouvelle fonction initSortables

```
function initSortables() {
    var options = {
        dropOnEmpty: true, containment: [ 'side1', 'side2' ],
        constraint: false, ghosting: true,
        onUpdate: function(list) {
            if (list.down('li'))
                list.removeClassName('empty');
            else
                list.addClassName('empty');
        }
    };
    Sortable.create('side1', options);
    Sortable.create('side2', options);
} // initSortables
```

Tant qu'une liste n'est pas vide, rien ne change par rapport à tout à l'heure. Mais si une liste est vide, on voit alors sa position, ce qui nous suggère qu'on peut déposer dessus à nouveau.

Figure 8-17
Une liste vide apparaît différemment.



Figure 8-18

On peut donc facilement y redéposer un élément.



Ah, ça fait plaisir !

Désactiver l'ordonnancement

Dans la tradition désormais établie de présence d'un moyen de désactivation, symétrique à l'activation de fonctionnalité, il est possible de faire qu'un conteneur cesse d'offrir l'ordonnancement par glisser-déplacer. Il suffit d'appeler `Sortable.destroy` sur le conteneur qu'on avait passé à `Sortable.create`. Plus simple, je ne vois pas...

Envoyer l'ordre au serveur

Enfin, modifier l'ordre des éléments ne servirait pas à grand-chose si on ne pouvait pas envoyer l'information au serveur pour qu'elle persiste. Vous vous en doutez, script.aculo.us n'a aucune intention de vous laisser patauger dans l'examen du DOM pour construire manuellement la liste des ID correspondant au nouvel ordre.

Vous avez peut-être remarqué que dans nos exemples, nous avons attribué à chaque élément contenu (en l'occurrence des `li`) un attribut `id` de la forme `préfixe_suffixe`. C'est une exigence de `Sortable.serialize`, la fonction que nous pouvons utiliser pour obtenir déjà une représentation URL encodée de l'ordre de nos éléments. Cette fonction produit une chaîne constituée de paramètres `préfixe[]` dont les valeurs sont les suffixes concernés, dans le bon ordre.

Par exemple, pour le DOM correspondant au HTML suivant :

```
<ul id="side1" class="people">
  <li id="person_1">Acy</li>
  <li id="person_2">Annabelle</li>
  <li id="person_3">Christophe</li>
  <li id="person_4">Élodie</li>
  <li id="person_5">Guillaume</li>
</ul>
```

Un appel à `Sortable.serialize('side1')` donnera ceci :

```
side1[] = 1 & side1[] = 2 & side1[] = 3 & side1[] = 4 & side1[] = 5
```

Cette syntaxe pour les noms résulte en un traitement automatique par la plupart des technologies côté serveur, PHP et Ruby On Rails pour ne citer qu'eux. En Java EE, on est un peu plus malheureux, comme toujours pour la gestion des paramètres : on devra utiliser un `getParameterValues('side1[]')` et itérer sur le résultat.

Notez que `serialize` accepte deux options.

Tableau 8-10 Options prises en charge par `Sortable.serialize`

Option	Description
<code>tag</code>	Identique à l'option homonyme de <code>Sortable.create</code> , et doit bien sûr être synchrone avec la valeur utilisée à l'appel de celle-ci.
<code>name</code>	Nom à utiliser pour les noms de champ de la représentation texte. Utilise par défaut l'ID du conteneur.

CompléTION AUTOMATIQUE DE TEXTE

Pour terminer avec les fonctions incontournables de script.aculo.us, penchons-nous sur ses possibilités de compléTION AUTOMATIQUE DE TEXTE. Il existe deux classes dédiées à ce domaine : `Ajax.AutoComplete` et `AutoCompleter.Local`. Nous nous intéresserons ici à la première, la seconde permettant une compléTION À PARTIR DE DONNÉES STOCKÉES CÔTÉ CLIENT, SOUS QUELQUE FORME QUE CE SOIT. Notez toutefois que les deux dérivent de `AutoCompleter.Base`, et partagent bon nombre de paramètres. Dans les tableaux 8-11 des options et 8-12 des fonctions de rappel ci-après, j'indique en italique les paramètres spécifiques à la variante Ajax.

CréATION D'UN CHAMP DE SAISIE À COMPLÉTION AUTOMATIQUE

Pour disposer d'un champ de saisie doté d'une compléTION AUTOMATIQUE DE TEXTE (champ généralement de type `<input type="text" ... />`), il faut en réalité définir

deux éléments dans votre HTML : le champ lui-même, plus un conteneur destiné à recevoir les suggestions. Il vous appartient de styler comme bon vous semble ce conteneur, pour être conforme à la charte graphique de votre site. En effet, la couche serveur doit y retourner une liste non ordonnée (`ul/li`), afin que `script.aculo.us` puisse correctement gérer les interactions clavier et souris.

Voici un exemple de bloc HTML :

```
<input type="text" id="edtName" name="contactName" />
<div id="nameCompletions"></div>
```

Notez la concision : c'est en partie dû au fait que `script.aculo.us` ajoutera pour vous un attribut `autocomplete="off"` au champ de saisie (deux types de complétion automatique rentreraient en conflit), et ajoutera si nécessaire une propriété CSS `position: absolute` à votre conteneur de suggestions.

Côté JavaScript, vous avez simplement besoin de construire un objet Ajax. `Autocompleter` autour de ces deux éléments, accompagné éventuellement d'options :

```
new Ajax.Autocompleter('edtName', 'nameCompletions',[, options]);
```

Voilà, c'est tout simple. Nous allons mettre en œuvre dans quelques instants deux exemples, l'un trivial, l'autre plus personnalisé. Mais avant, faisons le tour des options et fonctions de rappel disponibles.

Voici la liste des options prises en charge par notre `Ajax.Autocompleter`.

Tableau 8–11 Options prises en charge par `Ajax.Autocompleter`

Option	Description
<code>paramName</code>	Nom du paramètre à envoyer au serveur. Par défaut l'attribut <code>name</code> du champ de saisie.
<code>tokens</code>	Permet la complétion incrémentale. Voir plus bas. Par défaut un tableau vide : <code>[]</code> , donc désactivé.
<code>frequency</code>	Intervalle d'examen pour complétion, en secondes. Par défaut <code>0.4</code> . Évitez de descendre en dessous, pour des raisons d'ergonomie...
<code>minChars</code>	Nombre minimum de caractères à saisir avant de déclencher une complétion. <code>1</code> par défaut. <code>0</code> serait ignoré, mais mettre du négatif reviendrait à vous tirer dans le pied !
<code>indicator</code>	ID ou référence directe sur l'élément éventuel d'indication de progression (fortement conseillé), par exemple un <code>div</code> avec un <code>spinner</code> et un texte du style « assistance en cours... ». Rien par défaut.
<code>select</code>	Nom de classe servant à filtrer le contenu de l'option retenue pour extraire la valeur de complétion. Pas de valeur par défaut : l'ensemble de l'option est prise, à l'exception des contenus marqués avec une classe CSS <code>informal</code> (voir plus bas, « Personnalisation des contenus renvoyés »).

Tableau 8–11 Options prises en charge par Ajax.Autocompleter (suite)

Option	Description
autoSelect	Indique si lorsqu'un seul résultat est obtenu, il doit être automatiquement sélectionné. Vaut <code>false</code> par défaut.
parameters	Permet de définir des paramètres supplémentaires à envoyer au serveur pour compléter, afin d'ajouter généralement un contexte de recherche. Texte au format URL encodé. Vide par défaut.
asynchronous	Détermine si la requête est asynchrone (<code>true</code> , par défaut) ou non (<code>false</code>). Y toucher me semble une très mauvaise idée : une requête synchrone va geler le navigateur pendant son exécution...

Pour mémoire, seules les options en italique sont spécifiques à Ajax.Autocompleter : les autres viennent de Autocompleter.Base, et valent donc aussi pour Autocompleter.Local. Même chose pour les fonctions de rappel ci-dessous.

Il existe par ailleurs pas moins de cinq fonctions de rappel, qui sont toutefois très rarement utilisées, l'immense majorité des besoins étant déjà couverte par les options.

Tableau 8–12 Fonctions de rappel prises en charge par Ajax.Autocompleter

Fonction	Description
callback	Peut référencer une fonction chargée de modifier le texte du paramètre à envoyer au serveur, après composition initiale de celui-ci (<code>nom=valeur</code>). Il s'agit généralement d'ajouter un contexte fixe, ce qui se fait plus simplement avec <code>parameters</code> . Pas de valeur par défaut.
onShow	Référence la fonction appelée pour afficher la liste des suggestions. Par défaut, cale si besoin votre conteneur en position absolue immédiatement sous le champ de saisie, aligné en largeur, et le fait apparaître à l'aide d'un <code>Appear</code> de 1/8 de seconde.
onHide	Symétrique de <code>onShow</code> , qui par défaut masque le conteneur à l'aide d'un <code>Fade</code> de même durée.
updateElement	Appelée à la place de la fonction normalement chargée de placer dans le champ de saisie un texte correspondant à l'élément choisi dans la liste des suggestions.
afterUpdateElement	Appelée après l'insertion d'une valeur dans le champ de saisie suite à la validation d'une suggestion.

Dernier point avant d'aborder l'exemple, voyons comment script.aculo.us permet à l'utilisateur de faire son choix parmi les suggestions.

Interaction clavier et souris

À l'affichage du conteneur de suggestions, script.aculo.us sélectionne automatiquement la première, en lui ajoutant la classe CSS `selected` (à vous de styler comme

bon vous semble). La bibliothèque gère alors, tout le temps de l'affichage, les manipulations suivantes :

- survol de la souris, touches curseur (Haut, Bas, Gauche, Droite) : ajustement la sélection ;
- clic, Entrée, Tabulation : validation de la sélection ;
- Échap : fermeture du bloc des suggestions, retour à la frappe normale ;
- les autres frappes clavier s'ajoutent normalement à la saisie, et mettent donc à jour la liste des suggestions.

On a donc l'embarras du choix, l'utilisation du bloc de suggestion est plutôt accessible.

Attention : j'ai constaté un problème de capture de la touche Entrée (préférer Tabulation) sur Opera 9 et Safari 2.

Un premier exemple

Pour notre premier exemple, nous allons créer un formulaire de saisie d'un nom de bibliothèque Ruby. De cette façon, nous avons d'emblée à notre disposition l'ensemble des modules de la bibliothèque standard, fournis avec Ruby. Qui plus est, cela nous donnera l'occasion d'écrire un code serveur un peu plus puissant que d'habitude.

Copiez un de vos répertoires de travail récents, par exemple `draggable`, dans le sous-répertoire `docroot` d'un nouveau répertoire `autocomplete_simple`. Nous n'aurons pas besoin de `dragdrop.js`, mais il vous faudra ajouter le `controls.js` fourni dans l'archive de `script.aculo.us`. Voici déjà le fichier HTML.

Listing 8-37 Le HTML pour notre complétion simple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html";
    ↪ charset=iso-8859-15" />
  <title>Test simple de Ajax.Autocompleter</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="effects.js"></script>
  <script type="text/javascript" src="controls.js"></script>
  <script type="text/javascript" src="tests.js"></script>
</head>
<body>

<h1>Test simple de <code>Ajax.Autocompleter</code></h1>

<form>
```

```
<p>
  <label for="edtLibrary" accesskey="B">Bibliothèque Ruby</label>
  <input type="text" id="edtLibrary" name="library" />
  <div id="lib_suggestions" class="autocomplete"></div>
</p>
</form>

</body>
</html>
```

Remarquez la simplicité. En temps normal, notre formulaire aurait bien sûr une action, un bouton submit, etc. Passons au script client, qui reflète admirablement la simplicité de notre exemple.

Listing 8-38 Le script client, vraiment trivial !

```
function initAutocompleter() {
  new Ajax.Autocompleter('edtLibrary', 'lib_suggestions',
    '/getSuggestions');
} // initAutocompleter

document.observe('dom:loaded', initAutocompleter);
```

Trois paramètres suffisent : l'ID du champ de saisie, celui du conteneur de suggestions (qui sera automatiquement masqué à ce moment-là), et l'URL à invoquer (sans aucun paramètre additionnel, c'est important). Pour le reste, nous laissons l'ensemble des options à leur comportement par défaut.

Ceci suffirait, mais on obtiendrait un aspect visuel pas très professionnel, qui ressemblerait à ceci.

Figure 8-19
La complétion sans style particulier



C'est justement pour fournir un style passe-partout plus agréable que nous avons ajouté une classe CSS à notre `div` conteneur (afin de pouvoir réutiliser le style sur plusieurs champs à complétion dans la même page). Voici notre feuille de styles, qu'on pourrait encore simplifier, mais qui fait dans un certain raffinement.

Listing 8-39 Une feuille de styles passe-partout de qualité pour les suggestions

```
div.autocomplete {  
    position: absolute;          /* Histoire d'être explicite... */  
    width: 250px;                /* Sera ajusté par script.aculo.us */  
    background-color: white;  
    border: 1px solid #888;  
    margin: 0px;  
    padding: 0px;  
}  
  
div.autocomplete ul {  
    list-style-type: none;  
    margin: 0px;  
    padding: 0px;  
}  
  
div.autocomplete ul li {  
    list-style-type: none;  
    display: block;  
    margin: 0;  
    cursor: default;  
    /* Et quelques finasseries... */  
    padding: 0.1em 0.5ex;  
    font-family: sans-serif;  
    font-size: 90%;  
    color: #444;  
    height: 1.5em;  
    line-height: 1.5em;  
}  
  
/* Rappel : script.aculo.us active une classe 'selected'  
   lorsqu'un élément est sélectionné */  
div.autocomplete ul li.selected {  
    background-color: #ffb;  
}
```

À présent, passons à la couche serveur. Nous avons besoin de deux fichiers : notre éternel `serveur.rb` et un modèle pour la génération de la liste non ordonnée à renvoyer à la couche client. Voici déjà le modèle.

Listing 8-40 Le modèle trivial de réponse, suggestions.rhtml

```
<ul>
<% libs.each do |lib| %>
  <li><%= lib %></li>
<% end %>
</ul>
```

Et maintenant le cœur de l'exemple, la couche serveur...

Listing 8-41 La couche serveur pour la complétion automatique

```
#!/usr/bin/env ruby

require 'cgi'
require 'erb'
require 'webrick'
include WEBrick

template_text = File.read('suggestions.rhtml')
suggestions = ERB.new(template_text)

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

server.mount_proc('/get_suggestions') do |request, response|
  name_start = CGI::parse(request.body)['library'][0]
  suffix = "/#{Regexp.escape(name_start)}*.rb" ①
  libs = $LOAD_PATH.map { |dir| ②
    Dir.glob(dir + suffix, File::FNM_CASEFOLD).map { |f|
      File.basename(f, '.rb')
    }
  }.flatten.sort.uniq
  response['Content-Type'] = 'text/html'
  response.body = suggestions.result(binding)
end

trap('INT') { server.shutdown }

server.start
```

La ligne ① nous protège de fragments comme « * » ou « .. », qui n'auront pas d'effet particulier. À la ligne ②, \$LOAD_PATH est une variable globale en Ruby qui contient l'ensemble des répertoires où chercher les modules. Pour chacun, on y cherche des fichiers démarrant par notre texte et portant l'extension .rb, sans se soucier de la casse. On « aplatis » le tableau de tableaux obtenu, on le trie, et on retire les éventuels doublons.

Tout ça en si peu de lignes !

Voyons à présent le résultat, en quelques étapes.

Figure 8-20

Le champ à vide, en attente d'un premier caractère saisi



Figure 8-21

Complétion sur le premier caractère



Figure 8–22

Affinage avec
un deuxième caractère saisi

**Figure 8–23**

La touche Bas (ou la souris) nous amène sur la seconde suggestion.



N'est-ce pas sympathique comme tout, franchement ? Bon, quand vous aurez fini de tester toutes les lettres de l'alphabet pour voir les modules de la bibliothèque Ruby standard (j'ai beau les connaître, je n'ai moi-même pas pu m'en empêcher...), passez à la prochaine section pour découvrir comment renvoyer des contenus plus riches comme suggestions.

Figure 8-24

Le clic, Entrée ou Tabulation valident la suggestion.



Personnalisation des contenus renvoyés

En décrivant l'option `select`, nous avons simplement brossé la surface de la sélection de contenu opérée par les Autocompleters lorsqu'on choisit une des suggestions.

Quelle que soit la configuration, seuls les contenus textuels (en termes DOM, les nœuds texte) sont retenus, agrégés les uns aux autres avec une espace entre chacun. Le corollaire est immédiat : vous pouvez renvoyer, dans chaque `li`, un contenu structuré (`div`, `p`, `img`, etc.) : seules les portions textes seront récupérées. Ainsi, considérez l'élément suivant :

```
<li>
  
  Thomas Fuchs
  <div class="note">Auteur de script.aculo.us</div>
</li>
```

Sa sélection aboutirait à l'insertion du texte « Thomas Fuchs Auteur de script.aculo.us ».

Mais `script.aculo.us` ne s'arrête pas là. Il est fréquent de renvoyer des contenus textes plus riches que nécessaire, afin de fournir un contexte à l'utilisateur pour guider son choix. Par exemple, dans des suggestions de noms de contacts, on ajoutera l'adresse de courriel ou le nom de la société (voire une photo, mais comme ce n'est pas du texte, elle serait de toutes façons ignorée à la validation).

Le moyen le plus simple pour gérer cela consiste généralement à ajouter aux segments « superflus » une classe CSS `informal`. Par défaut, script.aculo.us ignore les éléments ainsi marqués. Du coup, en modifiant l'exemple précédent comme ceci :

```
<li>
  
  Thomas Fuchs
  <div class="note informal">Auteur de script.aculo.us</div>
</li>
```

On obtiendrait simplement en validant notre choix : `Thomas Fuchs`. Cependant, ce n'est parfois pas la façon idéale de travailler, notamment si vous avez beaucoup d'éléments superflus dans votre suggestion. C'est pourquoi l'option `select` vous permet de préciser une classe spécifique qui marque, au lieu des éléments à ignorer, les éléments à inclure.

Dans l'exemple qui va suivre, nous allons tirer parti de cette fonctionnalité pour renvoyer des contenus complexes mais ne retenir que le nom de chaque suggestion pour l'ajout dans le champ de saisie. Nous allons aussi illustrer la complétion incrémentale, qui permet de saisir plusieurs valeurs à la suite, en les séparant par un *token*, qui est généralement la virgule. On ne dérogera pas ici à la règle.

Copiez votre répertoire `autocomplete_simple` dans un nouveau répertoire `autocomplete_advanced`. La page HTML nécessite juste quelques ajustements cosmétiques (titre, pluriel au nom du champ et à son libellé, etc.).

Listing 8-42 Notre page HTML mise à jour pour une complétion incrémentale

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-15" />
  <title>Test avancé de Ajax.Autocompleter</title>
  <link rel="stylesheet" type="text/css" href="tests.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="tests.js"></script>
  <script type="text/javascript" src="controls.js"></script>
  <script type="text/javascript" src="effects.js"></script>
</head>
<body>

<h1>Test avancé de <code>Ajax.Autocompleter</code></h1>
```

```

<form>
  <p>
    <label for="edtLibraries" accesskey="B">Bibliothèques Ruby
    </label>
    <input type="text" id="edtLibraries" name="libraries" />
    <div id="lib_suggestions" class="autocomplete"></div>
  </p>
</form>

</body>
</html>

```

Le script est également mis à jour pour utiliser quelques options spéciales, afin notamment de continuer à envoyer un paramètre `library` alors que le champ, plus logiquement, s'appelle `libraries`.

Listing 8-43 Notre script ajusté

```

function initAutocompleter() {
  new Ajax.Autocompleter(
    'edtLibraries', 'lib_suggestions', '/getSuggestions',
    { paramName: 'library', tokens: [','], select: 'libName' });
} // initAutocompleter

document.observe('dom:loaded', initAutocompleter);

```

Pour chaque module Ruby, nous allons fournir sa taille et sa date de dernière modification. Nous aurons donc un modèle un peu plus riche.

Listing 8-44 Le modèle de suggestions avec les nouvelles données

```

<ul>
  <% libs.each do |lib| %>
    <li>
      <div class="libName"><%= lib.name %></div>
      <div class="libMTTime">
        ↪ <%= lib.mtime.strftime('%d/%m/%Y %H:%M:%S') %></div>
      <div class="libSize"><%= lib.size %> octets</div>
    </li>
  <% end %>
</ul>

```

Évidemment, la CSS suit.

Listing 8-45 La feuille de styles ajustée pour ces nouvelles données

```
div.autocomplete {  
    position: absolute;      /* Histoire d'être explicite... */  
    width: 250px;           /* Sera ajusté par script.aculo.us */  
    background-color: white;  
    border: 1px solid #888;  
    margin: 0px;  
    padding: 0px;  
}  
  
div.autocomplete ul {  
    list-style-type: none;  
    margin: 0px;  
    padding: 0px;  
}  
  
div.autocomplete ul li {  
    list-style-type: none;  
    display: block;  
    margin: 0;  
    cursor: default;  
    /* Et quelques finasseries (mais plus de color)... */  
    padding: 0.1em 0.5ex;  
    font-family: sans-serif;  
    font-size: 90%;  
    height: 3.5em;  
}  
  
/* Rappel : script.aculo.us active une classe 'selected'  
   lorsqu'un élément est sélectionné */  
div.autocomplete ul li.selected {  
    background-color: #ffb;  
}  
  
div.libMTtime, div.libSize {  
    font-size: 80%;  
    color: #444;  
}  
  
div.libMTtime {  
    margin: 0.2ex 0 0 2ex;  
}  
  
div.libName {  
    font-weight: bold;  
}
```

```
div.libSize {  
    margin: 0 0 0.5em 2ex;  
}
```

Remarquez la classe spéciale pour la portion dont le contenu nous intéresse (`libName`) lorsque l'utilisateur fera son choix. Bien, il ne nous reste plus qu'à mettre à jour la couche serveur.

Listing 8-46 Notre nouvelle couche serveur, avec une classe dédiée

```
#!/usr/bin/env ruby  
  
require 'cgi'  
require 'erb'  
require 'webrick'  
include WEBrick  
  
class LibInfo ❶  
    attr_reader :name, :mtime, :size  
  
    def initialize(name)  
        @name = File.basename(name, '.rb')  
        @mtime = File.mtime(name)  
        @size = File.size(name)  
    end  
  
    def <=>(other) ❷  
        self.name <=> other.name  
    end  
end  
  
template_text = File.read('suggestions.rhtml')  
suggestions = ERB.new(template_text)  
  
server = HTTPServer.new(:Port => 8042)  
server.mount('/', HTTPServlet::FileHandler, './docroot')  
  
server.mount_proc('/getSuggestions') do |request, response|  
    name_start = CGI::parse(request.body)['library'][0]  
    suffix = "#{$Regexp.escape(name_start)}*.rb"  
    libs = $LOAD_PATH.map { |dir|  
        Dir.glob(dir + suffix, File::FNM_CASEFOLD).map { |f|  
            LibInfo.new(f) ❸  
        }  
    }.flatten.sort.uniq  
    response['Content-Type'] = 'text/html'  
    response.body = suggestions.result(binding)  
end
```

```
| trap('INT') { server.shutdown }  
|  
server.start
```

- ➊ Petite classe dont les attributs sont en lecture seule, représentant les données que nous retenons pour chaque module. Le constructeur prend le nom complet du module (chemin compris) en argument.
- ➋ L'opérateur `<=>`, en Ruby, est utilisé pour les comparaisons, donc pour le tri. Ainsi, on préserve le fonctionnement de l'appel ultérieur à `sort` et `uniq`.
- ➌ Et voilà ! Au lieu d'utiliser directement le nom simple du module, on construit un objet `LibInfo`.

Observons à présent un exemple de saisie en plusieurs étapes.

Figure 8-25

Saisie d'un premier caractère et suggestions riches



Figure 8–26

Choix d'une première valeur

**Figure 8–27**

La validation n'a retenu que le nom.



Figure 8–28

Après avoir saisi le « token » (la virgule), on attaque une nouvelle valeur.

**Figure 8–29**

La validation ajoute la suggestion retenue, au lieu de remplacer.



Et ce n'est pas tout ! Il y a d'autres services

Désidément, script.aculo.us nous facilite beaucoup de choses. Et pourtant, nous n'avons vu qu'une partie du framework. Citons rapidement les services restants :

- *Slider* permet de réaliser des champs alternatifs de saisie de valeur numérique basés sur une rampe (verticale ou horizontale) sur laquelle on peut faire évoluer des poignées. Totalement portable, le système n'en est pas moins hautement configurable : poignées multiples, limitations dynamiques des intervalles de valeurs, etc.
 - Détails : <http://github.com/madrobby/scriptaculous/wikis/slider> (et vous trouverez des exemples beaucoup plus poussés dans les tests fonctionnels inclus dans l'archive).
- *In Place Editing* fournit deux mécanismes permettant de modifier à la volée une portion quelconque de la page et de synchroniser côté serveur (comme les titres et notes dans Netvibes, par exemple). Ce module a été entièrement réécrit vers la mi-2007. Hélas, la documentation en ligne n'a pas encore suivi (mais jetez un œil à la bibliographie en fin de chapitre).
- *Builder* simplifie grandement la création dynamique d'éléments *via* le DOM. On a bien vu, au chapitre 3, que la création DOM d'un contenu complexe est vite fastidieuse. *Builder* simplifie considérablement cette tâche : <http://github.com/madrobby/scriptaculous/wikis/builder>.
- *Sound* permet de lire des sons WAV ou MP3 sur plusieurs pistes sans recourir à Flash.

Avoir du recul : les cas où Ajax est une mauvaise idée

Toujours dans l'optique de qualité et de méthodologie de ce livre, il est temps d'apporter un regard critique sur les trésors de possibilités nouvelles que nous offrent ces frameworks (sans parler des nombreux outils prêts à l'emploi qu'on peut trouver dans les frameworks supplémentaires, évoqués dans l'annexe E).

Employer Ajax, des effets visuels ou des composants graphiques avancés n'est pas toujours une bonne idée : un usage à tout crin, sans considération ergonomique, peut rendre vos pages (et donc votre application web) inutilisable pour une partie significative des visiteurs et utilisateurs.

Nous allons d'abord évoquer l'ensemble des points à surveiller et des problèmes potentiels, pour finir par une série de pratiques recommandées qui permettent de les résoudre, ou au moins de les atténuer.

Ajax et l'accessibilité

On a déjà défini l'accessibilité dans la première partie de ce livre, en étudiant les relations entre celle-ci, JavaScript et le DOM. On l'a vu, l'accessibilité ne concerne pas que des utilisateurs non-voyants, comme on le croit souvent : elle concerne toute utilisation alternative de vos pages : périphériques à petit écran (de plus en plus répandus), sans clavier physique ou sans souris, handicaps visuels de toutes sortes (la cécité n'étant qu'un cas parmi d'autres), handicaps moteurs rendant l'utilisation du clavier ou de la souris difficile, handicaps cognitifs comme la dyslexie...

Nous avons vu que JavaScript, et les pages dynamiques en général, ne vont pas forcément à l'encontre d'une accessibilité soignée. Mais il est évident que l'emploi de mécanismes encore plus dynamiques (effets visuels, glisser-déplacer, etc.) et le recours à Ajax soulèvent de nouvelles questions quant à l'accessibilité et l'ergonomie.

Considérations techniques

Nous avons d'abord la question de la disponibilité de XMLHttpRequest. Dans l'état actuel des navigateurs, disposer de XMLHttpRequest revient à avoir JavaScript activé et le DOM niveau 2 pris en charge, ce qui couvre la quasi-totalité des navigateurs répandus. Il existe tout de même un problème : le fait que sur MSIE 6 et antérieurs, XMLHttpRequest est fourni comme un ActiveX.

En effet, en raison de la très faible sécurité assurée par cette technologie, de très nombreux administrateurs système en entreprise désactivent purement et simplement ActiveX sur l'ensemble des MSIE de leur parc informatique. Sur ce type de poste client, même si nos pages peuvent utiliser JavaScript et manipuler le DOM, elles seront incapables d'utiliser Ajax. Ce n'est pas forcément un problème pour les applications intranet, ces mêmes responsables informatiques mettant généralement en place une politique plus tolérante pour la « zone intranet » reconnue par MSIE.

Le problème se corse encore davantage lorsque le service informatique recourt à une mesure encore plus draconienne, fréquente pour un parc où certains postes ne peuvent pas être verrouillés, et risquent donc d'avoir ActiveX activé : le filtrage au niveau du serveur mandataire (*proxy*). Ce filtrage passe généralement à la trappe tout fichier HTML ou JavaScript contenant le texte `new ActiveXObject`, ce qui a pour effet de supprimer le script entier ! Une portion significative, voire la totalité de votre logique applicative côté client est ainsi inutilisable.

Les lecteurs d'écran constituent également un point douloureux. Par essence, employer Ajax revient à mettre à jour une portion de la page plutôt que l'ensemble de la page. Or, si l'ensemble des études et tests menés ces deux dernières années montre quelque chose, c'est bien que les lecteurs d'écran sont encore très rudimentaires pour tout ce qui touche aux pages dynamiques.

Outre une pléthore de comportements dénués d'un quelconque effort de standardisation, pas un lecteur d'écran disponible aujourd'hui ne réagit correctement à un changement soudain d'une portion de la page, quand bien même cette portion serait à ce moment en cours de lecture, et en dépit de nombreux efforts dans le script visant à aider le logiciel à s'y retrouver (on peut par exemple consulter les tests édifiants menés en mai 2006 par James Edwards et d'autres sommités de l'accessibilité sur les sept principaux lecteurs du marché, sur <http://www.sitepoint.com/article/ajax-screenreaders-work>).

Considérations ergonomiques

Et quand bien même tout va bien techniquement, la plus grande source de problèmes se trouve probablement au niveau de l'ergonomie des sites. D'ailleurs, faut-il parler de sites ou d'applications ? La nuance peut sembler futile, mais elle joue un grand rôle dans l'esprit de l'utilisateur et dans ses attentes quant à l'ergonomie que vous proposez.

La plupart des internautes ont une association encore bien ancrée dans leur esprit :
navigateur = pages = site = interaction faiblarde

Par opposition, on a aussi :

application = écrans = programme (local) = interaction riche

Le terme application web (ou sa contraction *webapp*) ne suffisait pas à mélanger ces perceptions jusqu'à récemment, puisque les applications en question conservaient le schéma classique requête/chargement/réponse. Ajax et les frameworks JavaScript récents ont donné lieu à une nouvelle génération d'applications web, pour lesquelles un acronyme existe déjà : RIA, *Rich Internet Applications*, terme sur lequel se concentrent de plus en plus les efforts marketing du Web 2.0. Il est vrai que Flash permettait de réaliser des interfaces riches depuis longtemps, mais de façon propriétaire et, jusqu'à récemment, plutôt restreinte en termes de communications avec le serveur (Flex est en train de modifier cet état de choses, ceci dit). Flash est aussi souvent plus lourd et moins accessible que l'alternative Ajax.

Quoi qu'il en soit, tant que vos utilisateurs ne penseront pas à vos pages comme à une véritable application (et cela prendra du temps), ils n'auront pas les mêmes attentes. Là où il leur semble naturel, par exemple, de faire glisser une icône de document sur celle de la poubelle de leur bureau, ce même comportement ne leur vient pour l'instant pas à l'esprit lorsqu'ils sont dans un navigateur.

Ce hiatus peut poser plusieurs problèmes :

- Lorsque vous déclenchez une requête Ajax en arrière-plan, l'utilisateur ne perçoit pas forcément qu'il y a une requête en cours... Du coup, il peut être tenté d'utiliser l'interface de façon inappropriée (quitter la page ou la recharger, effectuer une action ailleurs sur la page qui risque d'entrer en conflit avec le traitement du résultat de la première requête, etc.).

- Dans la même série, l'utilisateur ne s'attend pas à pouvoir utiliser des mécanismes comme le glisser-déplacer dans une page web, à moins qu'on rende cette possibilité évidente d'une façon ou d'une autre.
- Les changements très localisés sur la page ne sont pas forcément remarqués. L'utilisateur peut regarder son clavier en tapant ou avoir l'habitude plus ou moins consciente de guetter le rechargement de la page comme signe que le traitement a eu lieu : faute de recharge visible, il ou elle n'aura pas forcément le réflexe d'examiner à nouveau le contenu.

Utilisations pertinentes et non pertinentes

Indépendamment des attentes de l'utilisateur, il existe des contextes dans lesquels utiliser Ajax est tout simplement contre-intuitif, voire contre-productif, ou peut avoir des effets de bord indésirables.

Ainsi, on a deux types de pages dans une application web : les pages de transition, dont l'état n'a pas vocation à persister, et les autres. Avant Ajax, chaque étape d'un processus, qu'elle représente un état transitoire ou un état stable (qu'on pourrait vouloir ajouter à ses marque-pages ou dont on souhaiterait envoyer l'URL à un ami), était obtenue à l'aide d'un chargement de page.

Le bouton *Précédent* (ou la touche associée), bien connu des internautes, et très utilisé notamment par les utilisateurs non avancés, fonctionnait alors conformément à l'intuition : il nous ramenait à l'étape précédente (peu importe que l'application s'en offusque ou pas, ce n'est pas le sujet).

Mais si cette série d'étapes passe à Ajax, il n'y aura pas de recharge de la page d'une étape à l'autre : l'entrée précédente dans l'historique de navigation reste la page visitée avant de démarrer le processus en question. Cela peut causer une confusion chez l'utilisateur, qui va naturellement cliquer *Précédent* s'il souhaite revenir à l'étape précédente, pour se retrouver en réalité plusieurs pages en amont dans sa navigation ! Même s'il existe des astuces JavaScript qui permettent de masquer ce problème, ce sont un peu des « rustines »...

Il convient donc de bien examiner si les différentes étapes constituent un état essentiellement fugitif (par exemple des résultats de recherche, la saisie d'un commentaire dans un blog, une validation de données), auquel cas Ajax est parfaitement acceptable, ou s'il ne serait pas préférable de recourir à une navigation plus traditionnelle.

Cette distinction intervient aussi au niveau des URL des pages : s'il perçoit la vue courante comme quelque chose de stable, qui pourra être consultée ultérieurement, l'utilisateur peut vouloir en utiliser l'URL (pour un marque-page, pour l'envoyer à un ami, etc.). Dans un système Ajax, l'URL ne change évidemment pas tout au long des

étapes, et l'utiliser plus tard nous amènera au début du processus, ce qui sera là aussi source de confusion.

Enfin, gardez à l'esprit que pour la majorité des formulaires (inscription, modification de profil, etc.) leur envoi manuel, à l'aide d'un traditionnel champ de type `submit` ou `image`, est parfaitement justifié : faire passer ces formulaires en envoi Ajax à la volée empêcherait l'utilisateur de les explorer, de se faire une idée de sa saisie, ou d'interrompre celle-ci pour la compléter ou la reprendre ultérieurement. Dans la mesure où, la plupart du temps, vous ne fournissez pas de mécanisme d'annulation par la suite, un envoi forcé à la saisie serait perçu comme trop intrusif, voire sérieusement gênant.

Pratiques recommandées

Voici l'essentiel des recommandations pertinentes qu'on trouve aujourd'hui au sujet de l'emploi d'Ajax dans les applications web. Pour le reste, deux règles d'or : faire preuve de bon sens et se mettre réellement à la place de l'utilisateur novice, qui découvre vos pages sans aucun *a priori* sur leur utilisation.

Principes généraux

Une règle de base, qui vaut pour toutes les pages, est de commencer par un fonctionnement traditionnel, sans JavaScript, reposant uniquement sur les éléments actifs habituels : liens et formulaires. Il ne faut pas pour autant perdre de vue qu'on va ensuite enrichir ce fonctionnement avec JavaScript et de l'Ajax, afin d'éviter notamment des erreurs de modularisation côté serveur. Mais en partant d'une conception 100 % passive côté client, pour ensuite l'enrichir avec de l'*Unobtrusive JavaScript*, on assure automatiquement une dégradation élégante.

Cette approche d'amélioration progressive (le terme anglais *progressive enhancement* a fait école) a trouvé un nom bien à elle, lorsqu'elle s'applique à l'ajout d'un fonctionnement Ajax : Jeremy Keith, figure de proue de JavaScript et du DOM, l'a baptisée *Hijax* (<http://www.domscripting.com/blog/display/41>). Je ne saurais trop vous recommander de procéder ainsi : le surcoût en temps est minime, et les bénéfices sont énormes.

Par exemple, l'application en ligne Backpack de 37signals (<http://backpackit.com>), en dépit d'une interface très riche et exploitant largement JavaScript, le DOM et Ajax, reste parfaitement utilisable sur des navigateurs textuels comme Lynx. Gmail, en revanche, est d'une accessibilité très discutable (<http://www.chaddickerson.com/blog/2005/10/18>, puis choisissez l'unique article).

Autre point général important : sur un site public, pensez à bien spécifier dès la page d'accueil que vous utilisez JavaScript et Ajax, et faites un petit exposé sur les impacts

ergonomiques. Cela aide grandement à ajuster convenablement les attentes de vos utilisateurs, et améliore leur ressenti sur votre application.

Ergonomie et attentes de l'utilisateur

- Pendant une requête Ajax, signifiez visuellement, de façon claire, qu'un traitement est en cours. Si certaines portions de l'interface peuvent poser problème pendant ce temps-là, désactivez-les, comme nous l'avons fait dans notre exemple de réponse XML au chapitre 6.
- Si vous avez recours au glisser-déplacer, rendez la possibilité évidente, aussi claire que possible, et de préférence à la fois textuellement et graphiquement. Et bien entendu, il doit y avoir une façon alternative, accessible, d'obtenir le même résultat (de nombreux périphériques ne permettront pas de simuler le glisser-déplacer, et les personnes souffrant d'un handicap moteur rendant l'usage de la souris difficile auront le plus grand mal à utiliser le glisser-déplacer).
- Lorsque vous venez d'ajouter ou de modifier un élément de la page, mettez-le en exergue visuelle, par exemple avec un effet *Highlight* ou *Pulsate* (comme dans notre dernier exemple d'ajout de commentaires). Au cas où la partie mise à jour ne serait pas à l'écran (en raison d'un défilement, ou d'une bascule de l'utilisateur sur une autre application pendant le traitement), il existe aussi des techniques pour émettre un son. Par exemple, Campfire, une application de forums interactifs en ligne (<http://campfirenow.com>), émet un son discret mais reconnaissable lorsqu'un nouveau message arrive.
- N'utilisez jamais Ajax pour envoyer une saisie automatiquement, à moins que cela ne soit le comportement général approprié à votre application (par exemple, dans le cas de Netvibes), ou qu'il ne s'agisse que d'une sauvegarde temporaire permettant une reprise ultérieure (à la Web Forms 2.0) ou une résistance à la panne (par exemple, dans un passage de QCM en ligne pour un examen).
- Bien que je recommande plutôt de recourir à une navigation traditionnelle quand le besoin s'en fait sentir, il existe un moyen de gérer le bouton Précédent et de fournir des URL adaptées à l'état dans le cadre d'états temporaires de pages utilisant Ajax. Voyez l'article de Mike Stenhouse : <http://www.contentwithstyle.co.uk/Articles/38>
- Si vous souhaitez fournir à vos utilisateurs la possibilité d'accéder ultérieurement à un état de votre page qu'ils ont obtenu par manipulation Ajax, vous pouvez tout à fait créer votre propre solution, avec l'ensemble des paramètres nécessaires dans l'URL, comme le fait par exemple Google Maps avec son lien « Obtenir l'URL de cette page », en haut à droite de la carte. Il vous appartient alors de mettre cette fonction bien en évidence.

Cognitif/Lecteurs d'écran

- Tant pour les utilisateurs ayant des difficultés de concentration que pour ceux utilisant un lecteur d'écran, il est préférable de déclencher une requête Ajax manuellement, afin d'être prêt à examiner son résultat, que d'avoir affaire à des modifications périodiques dont on ne décèle pas toujours l'exécution.
Si vous utilisez un mécanisme de type `PeriodicalUpdater`, il est alors intelligent de fournir une option, facilement accessible, permettant à l'utilisateur d'opter pour un déclenchement manuel (un bouton ou un lien couplé à un `Updater` classique, par exemple). Le choix pourrait être proposé en début de page à l'aide d'une case à cocher, par exemple.
- Pour les traitements Ajax consécutifs à la frappe (complétion automatique, etc.) il vaut mieux éviter de déclencher la requête immédiatement : il est préférable d'attendre un bref intervalle d'inactivité, afin de laisser l'utilisateur taper aussi loin qu'il le souhaite, à sa vitesse de frappe normale. En effet, en l'interrompant à chaque caractère ou presque avec une liste de suggestions, on ne réussit qu'à détourner sans cesse son attention et à augmenter, au final, son temps de saisie.
- Il peut être intéressant aussi de permettre à l'utilisateur d'être explicitement averti d'une mise à jour d'une portion de la page suite à un traitement Ajax, par exemple à l'aide d'un message d'avertissement (`alert`). Cette technique présente l'avantage d'amener automatiquement l'attention de l'utilisateur et le focus de l'éventuel lecteur d'écran sur la notification, tout en restaurant ensuite le focus de ce même lecteur d'écran là où l'utilisateur lisait auparavant.
Pensez à rendre le message explicite, avec un contexte clair : préférez « Le nombre d'inscrits est désormais de 430 » ou « Votre commentaire est désormais présent en fin de liste » à « Opération réussie »...
Enfin, cela doit bien sûr ne rester qu'une option, car cela gênerait les utilisateurs n'en ayant pas besoin. Par ailleurs, il ne s'agit pas d'une panacée : certains lecteurs d'écran ne liront pas automatiquement le texte du message !
- Dans le cas précis des lecteurs d'écran, dans la mesure où, on l'a vu, certains gèrent à peu près JavaScript, mais aucun ne réagit correctement à une modification partielle due à Ajax, la meilleure solution peut être d'offrir à l'utilisateur la possibilité de désactiver purement et simplement Ajax. Il ne s'agit pas de lui demander de manipuler les options du navigateur, mais bien de cocher par exemple une case, qui ajustera le fonctionnement de vos scripts et basculera donc sur une navigation traditionnelle. Pour peu que vous ayez développé façon *Hijax*, comme vu plus haut, ce sera facile à implémenter.

Pour conclure, voici quelques articles particulièrement intéressants sur l'implémentation d'une utilisation dégradable d'Ajax et l'utilisabilité des formulaires qui y ont recours :

- <http://adactio.com/journal/959>
- <http://particletree.com/features/the-hows-and-whys-of-degradable-ajax/>
- <http://particletree.com/features/degradable-ajax-form-validation/>
- <http://www.baekdal.com/articles/usability/usable-XMLHttpRequest/>

Pour aller plus loin...

Site

Je l'ai cité très souvent, le site officiel de script.aculo.us contient de très nombreuses documentations et exemples : <http://script.aculo.us>

Groupe de discussion

Le groupe Google RubyOnRails-Spinoffs a été créé en août 2006 pour servir de centre d'aide technique, avec les avantages offerts par les outils d'indexation et de recherche de Google. On y trouve de nombreux membres ayant un bon niveau technique. Hélas, il a fini par crouler sous le spam faute d'une modération active.

Heureusement, à l'initiative de TJ. Crowder, un groupe de remplacement, au nom d'ailleurs plus explicite, a vu le jour en juillet 2008 et concentre désormais toute l'activité, le groupe historique n'existant plus que pour ses archives : <http://groups.google.com/group/prototype-scriptaculous>.

Canal IRC

Enfin, il faut noter qu'on trouve souvent une réponse rapide et fiable sur le canal IRC dédié à script.aculo.us, hébergé sur l'incontournable serveur `irc.freenode.net`. Le canal se nomme tout simplement `#scriptaculous`.

TROISIÈME PARTIE

Interagir avec le reste du monde

Pour découvrir et maîtriser Ajax, nous avons eu recours à une couche serveur locale, facile à personnaliser pour nos tests. Dans le développement de vos applications, vous allez généralement utiliser un service qui vous est propre, hébergé sur vos serveurs : c'est une situation similaire.

Et pourtant, la puissance du Web 2.0 tient aussi largement à l'interopérabilité des services, et à la possibilité pour une page de recouper des informations d'origines diverses et d'exploiter des services fournis par des tiers.

On s'intéressera à deux grandes catégories d'interaction.

Dans le chapitre 9, nous examinerons les fameux Web Services, spécifiquement ceux proposés au travers d'interfaces REST. Ces derniers sont de plus en plus fréquents, et prennent petit à petit le pas sur l'univers SOAP / WS-*, souvent jugé trop complexe et trop lourd.

Le chapitre 10 se penchera sur la syndication de contenus, principalement au travers des deux formats de flux dominants : RSS 2.0 et Atom 1.0.

Le chapitre 11 complétera ce tour d'horizon en exploitant certaines API 100 % JavaScript, ne nécessitant donc aucun traitement sur votre serveur, pour incorporer des services externes (cartographie, graphiques...) comme le veut la tendance actuelle aux « *mashups* ».

Grâce à ces outils, vous pourrez rendre vos pages plus riches en contenu, en possibilités, en intérêt. Et peut-être, si ce n'est déjà fait, réaliser que dans le Web 2.0, l'interopérabilité n'est pas un vain mot.

9

Services web et REST : nous ne sommes plus seuls

Les services web étaient censés permettre à n'importe quel logiciel de communiquer par Internet avec n'importe quel autre, quels que soient leurs langages de programmation respectifs.

La promesse n'est que péniblement tenue, au point qu'une sorte de retour aux sources s'est opéré, qui a découvert des trésors dans ce bon vieux protocole HTTP, et ne voudra plus nécessairement un culte à XML. On fait maintenant des API REST et le capharnaüm de SOAP et WS-* est déjà sur le chemin de la retraite, en dépit de quelques tentatives acharnées.

Nous verrons dans ce chapitre à quoi correspondent ces technologies. Après avoir aperçu les horizons qu'elles nous ouvrent, nous réaliserons quelques exemples concrets au travers d'API REST pour discuter avec Amazon.fr, The Weather Channel et Flickr.

Pourquoi la page ne parlerait-elle qu'à son propre site ?

Tout simplement pour des raisons techniques. Conceptuellement, il est parfaitement acceptable que votre page fédère du contenu qu'elle obtient dynamiquement depuis des services hébergés sur d'autres serveurs. C'est fondamentalement le rôle des services web.

Du temps des services web « classiques » (avec leurs deux tonnes nominales de SOAP pour dire bonjour), personne n'aurait osé imaginer que les communications allaient avoir lieu directement depuis la couche client. Ce serait pourtant génial, car cela déchargerait votre serveur. Cependant, SOAP est tellement complexe que c'était inimaginable, à moins de recourir à Flash, à une applet Java ou, sur MSIE, à un ActiveX (au prix d'une damnation éternelle).

Imaginez un peu les possibilités... Aujourd'hui, tout service digne de ce nom fournit des services web en mettant le plus souvent en avant une API REST (les services très à la pointe n'offrent d'ailleurs que REST). Regardez Netvibes (qui fait déjà figure de vétéran...) : tout le contenu de la page et tous les services viennent de l'extérieur. Des pages de portail personnalisé, comme Google IG, sont entièrement basées sur ce concept.

Contraintes de sécurité sur le navigateur

Seulement voilà, les possibilités, vous n'êtes pas les seuls à les imaginer. Des personnes aux intentions moins honorables pourraient s'en servir pour contourner des mesures classiques de sécurité.

Imaginons par exemple que vous êtes sur votre poste en entreprise. Vous avez accès à l'intranet, lequel contient sans doute des informations confidentielles. Vous avez aussi accès à Internet, bien entendu. Et là, vous accédez à une page dotée d'un script qui tente de récupérer des informations sur `http://intranet/...` pour ensuite les transmettre à un site distant. Vous imaginez la suite !

C'est pour cette raison que sur la plupart des navigateurs, JavaScript interdit l'accès à d'autres domaines que celui dont la page est issue. Il peut s'agir d'un script issu d'un domaine A tentant de manipuler les fonctions ou variables d'un autre script (fourni par un `frame` ou un `iframe`) issu d'un domaine B. Dans le cas qui nous occupe ici, il s'agit d'un objet `XMLHttpRequest` qui n'a pas le droit de communiquer avec un domaine tiers.

Suivant le navigateur, cette contrainte est plus ou moins prononcée. Il est souvent possible d'accéder à un *domaine parent*. Par exemple, un script exécuté par une page de `intranet.example.com` peut avoir accès aux ressources de `example.com`. C'est la fameuse exception à la *Same Origin Policy*, qui remonte à Netscape Navigator 2.0 (ce qui ne nous rajeunit pas). On utilise pour cela l'attribut DOM propriétaire

`document.domain`. Vous trouverez une explication plus détaillée de la S.O.P. à la page suivante : <http://www.mozilla.org/projects/security/components/same-origin.html>.

Certains navigateurs ne mettent en place aucune restriction, par exemple MSIE (qui n'est pas à une faille de sécurité près, il faut dire).

En revanche, les navigateurs Mozilla (Mozilla, Firefox et Camino) sont très stricts. Pour désactiver cette limitation, il faut fournir le script en tant que **script signé**. Il est aussi possible de configurer spécialement votre navigateur de développement pour éviter cela pendant les tests (mais cela ne remplace pas la signature pour le déploiement en production).

Vous trouverez tous les détails sur <http://developer.mozilla.org/en/docs/Security>, mais sincèrement, le jeu n'en vaut pas la chandelle : en production, cela ne marchera pas, en tout cas pas assez pour XMLHttpRequest. L'internaute qui utilise vos pages n'a probablement pas effectué ces réglages (il n'en a peut-être pas besoin s'il utilise un navigateur moins sécurisé). Le seul moyen de fournir une page utilisant ces droits étendus à un tiers inconnu consiste à fournir la page et ses scripts comme une **archive signée**.

Outre le fait que cela nécessite un certificat apte à signer du code (ce n'est pas toujours le cas des certificats utilisés pour HTTPS) et qu'il faut ensuite utiliser un outil spécifique (SignTool) à chaque mise à jour du contenu de l'archive, cela change radicalement la façon d'invoquer la page depuis le reste du site (l'URL a l'aspect suivant : `jar:http://www.example.com/truc.jar!/page.html`). Du coup, c'est incompatible avec les autres navigateurs...

Notez bien qu'une spécification est en cours de mise au point pour permettre l'utilisation sécurisée de requêtes Ajax à travers plusieurs domaines, principalement au moyen d'en-têtes de réponse HTTP qui indiqueront explicitement les « domaines cibles » et « domaines sources » autorisés. C'est une voie prometteuse qui pourrait résoudre bien des aspects du problème, mais à l'heure où j'écris ces lignes, cet effort n'a pas encore abouti.

Une « couche proxy » sur votre serveur

La seule solution véritablement portable, qui a de surcroît l'avantage de ne rien demander à l'utilisateur, consiste à mettre en place une couche intermédiaire sur votre serveur, qui se contente d'effectuer la requête qu'on lui donne, et de transmettre le résultat tel quel à votre script. C'est ce procédé que nous utiliserons au cours de ce chapitre. En effet, notre couche serveur n'est pas soumise aux contraintes de sécurité du navigateur : elle peut émettre des requêtes où bon lui semble (tant que les routeurs laissent passer, en tout cas).

Il existe tout de même une différence entre l'utilisation d'une simple fonction de proxy et le modèle traditionnel de traitement sur la couche serveur. Ici, le traitement est fait côté client : c'est plus léger pour le serveur, qui n'est qu'entremetteur, et cela ouvre des possibilités d'extensions pour vos pages, sur le même principe que les extensions Firefox.

Architecture de nos exemples

Nos trois exemples pour ce chapitre suivront tous le même principe, mais avec divers degrés de complexité et de raffinement. L'architecture fondamentale est la suivante :

- Notre page a besoin d'un document XML émis par un service tiers.
- Ne pouvant y accéder directement, elle passe par une fonction « proxy » proposée par notre couche serveur. C'est vers cette fonction que notre requête Ajax s'effectue.
- Le document obtenu est généralement trop complexe pour une extraction manuelle à l'aide des propriétés DOM ; on passera donc soit par une extraction XPath (comme nous l'avons déjà fait au chapitre 6), soit directement par une feuille de styles XSLT, pour obtenir un fragment XHTML représentant les informations qui nous intéressent.

Peut-être n'avez-vous jamais eu l'occasion d'utiliser XSL ou XSLT ? L'abréviation XSL signifie *eXtensible Stylesheet Language*. C'est la principale technologie de mise en forme d'un contenu XML, avec CSS. Cependant, là où CSS ne fait que réaliser une mise en forme à destination d'un médium (imprimante, écran, vidéo-projection), XSL ouvre beaucoup plus de possibilités.

On considère traditionnellement que XSL est composé de deux technologies : XSL-FO (*Formatting Objects*, finalisée en 2001), une architecture permettant de réaliser des adaptateurs pour des formats spécifiques (le cas le plus courant étant PDF), et XSLT (*Transform*, finalisée en 1999), qui transforme le document source en un document quelconque, le plus souvent balisé lui aussi (XHTML, SVG, MathML...), bien que rien ne l'exige. Dans les deux cas, XSL utilise XPath pour extraire les données souhaitées du document XML source.

Afin de réaliser des extractions XPath et des transformations XSLT directement dans nos pages web, nous allons recourir encore une fois à Google AJAXSLT. En effet, la prise en charge de ces technologies par les navigateurs, et surtout leur accessibilité par JavaScript, sont très hétéroclites.

Vous trouverez davantage d'informations sur XSL, XSL-FO et XSLT aux adresses web citées en fin de chapitre. Les feuilles XSLT utilisées dans ce chapitre ne sont pas très compliquées.

Comme toujours, les exemples complets sont dans l'archive des codes source.

Comprendre les services web

Le terme « service web » est apparu il y a quelques années pour désigner un mécanisme « portable » de communication entre objets distants, une catégorie de logiciels appelés ORB (*Object Request Broker*).

Les technologies ORB traditionnelles, CORBA, DCOM et RMI, se révélaient en effet insatisfaisantes pour la majorité des besoins. On leur reprochait en effet souvent les points suivants :

- Elles sont attachées à certains langages ou à des plates-formes particulières, en dépit d'efforts ultérieurs de portabilité.
- Elles utilisent un format propriétaire et binaire de transfert (débogage complexe).
- Elles utilisent le réseau d'une façon parfois trop complexe ou trop lourde à gérer (ports propriétaires nécessitant des configurations de routeurs, maintien de connexion et maintien d'état, etc.).
- Elles nécessitent la génération, souvent statique (à la compilation, pas à l'exécution), de classes intermédiaires (*skeletons* et *stubs*).
- Elles nécessitent un référentiel centralisé des objets disponibles, ainsi parfois qu'un référentiel des objets actifs.

En dépit des performances souvent élevées offertes par ces technologies, leur mise en œuvre était jugée trop complexe pour de nombreux cas, et surtout recelait toujours des difficultés de portabilité et de maintenance.

Les services web devaient initialement reposer sur un format et un protocole ouverts, stables, robustes et bien maîtrisés : XML et HTTP. On a aussi souhaité dès le départ éviter toute gestion d'état au niveau du protocole, afin de conserver les principaux avantages de HTTP : la capacité à monter facilement en charge, et la tolérance à la panne courte. Un format basé sur XML devait permettre la description d'un service : WSDL (*Web Service Description Language*). Le protocole dédié aux services devait donc être simple, portable, et facile à mettre en œuvre. On le baptisa SOAP, pour *Simple Object Access Protocol*.

Hélas, l'univers des standards WS (*Web Services*) a tellement enflé, et l'effet *Design By Committee* a généré une telle complexité, que cet acronyme est devenu une vaste farce. D'ailleurs, la version 1.2 de SOAP l'a officiellement retiré. SOAP est devenu SOAP, qui n'est plus un acronyme. Le protocole est en effet devenu atrocement complexe (avec l'ajout de possibilités d'utilisation de SMTP ou XMPP au lieu de HTTP, les pièces jointes, etc.), et entouré d'une pléthore d'autres formats et protocoles : UDDI pour les référentiels de services, et les WS-*, de plus en plus honnis, qui ne cessent de fleurir (WS-Addressing, WS-Security, WS-Reliable-Exchange, etc.). Le W3C compte six groupes de travail distincts sur le sujet !

Le résultat d'une telle surenchère est un schisme classique. D'un côté, on trouve les poids lourds de l'industrie, qui ont consacré des millions au développement de produits et services basés sur ces technologies, et les poussent en avant afin de récupérer leur investissement (sur ce point, les courriels circulant dans les listes de diffusion des membres du W3C sont édifiants). Dans le même camp, on trouve tous les projets à gros budget dont les directeurs techniques ont succombé au chant de sirène des poids lourds en question.

De l'autre côté, on a, comme d'habitude, le camp des partisans d'un développement agile, rapide, flexible, souple, et néanmoins performant. C'est le camp de ceux qui ont mis de côté XHTML 2.0 et XForms au profit des alternatives pragmatiques à l'origine du groupe de travail HTML 5 : principalement Web Applications 1.0 (<http://whatwg.org/specs/web-apps/current-work/>) et Web Forms 2.0 (<http://whatwg.org/specs/web-forms/current-work/>) ; le camp de ceux qui préfèrent Ruby On Rails à l'univers J2EE... le camp, enfin, qui a proposé REST comme alternative au dogme « WS-* ».

Qu'est-ce qu'une API REST ?

REST est l'acronyme de *REpresentational State Transfer*. Ce terme en soi est particulièrement malheureux, car il ne renseigne en rien sur la nature concrète de l'approche. Fournir une API REST sert le même objectif que fournir une API SOAP : permettre à des codes externes de dialoguer avec notre service, de façon portable et robuste. Dans les deux cas, on se repose sur HTTP. Cependant, la similitude s'arrête ici.

Dans une API REST, l'application est exposée sous forme de **ressources** : un gestionnaire d'utilisateurs, un compte utilisateur, un tableau de bord, un mémo, une tâche à faire, une liste des fichiers joints, etc. Chaque ressource est accessible au travers d'une URL unique, utilisée pour tout : liens vers la page concernée, demande de modification, de suppression, etc.

Chaque ressource fournit une interface d'accès uniforme, constituée principalement de deux aspects :

- une série d'opérations correspondant à des verbes HTTP (principalement POST, GET, PUT et DELETE, qui correspondent aux opérations de création, de lecture, de mise à jour et de suppression, ce qu'on appelle classiquement CRUD : *Create, Read, Update, Delete*) ;
- une série de types de contenus MIME, en requête comme en réponse. Ceux en requête offrent juste plus de souplesse quant au format de transfert des paramètres, tandis que les types de réponse attendus peuvent changer radicalement la *nature* de la réponse fournie par la ressource.

REST constitue une approche client/serveur sans gestion d'état (en tout cas, pas par le protocole : comme en HTTP classique, l'utilisation de cookies ou d'un équivalent peut simuler l'état). Sa simplicité permet aussi l'utilisation pertinente d'un système de cache, ou l'enrobage dans des couches supplémentaires (par exemple une couche d'authentification).

L'avantage majeur de REST est sa simplicité, et la facilité avec laquelle une application web classique peut fournir une API REST rien qu'en prenant en charge quelques verbes HTTP supplémentaires dans sa couche serveur lorsque c'est pertinent.

Il est tout simplement impensable d'avoir un client SOAP en JavaScript : le code résultant serait bien trop lourd ; il faudrait recourir à un objet natif non standard, une applet Java ou, sur MSIE, un contrôle ActiveX.

En revanche, avoir un client REST est trivial avec XMLHttpRequest, au moins pour les verbes GET et POST. Les verbes suivants ne sont pas toujours pris en charge, mais le standard W3C en cours de finalisation les prévoit explicitement.

REST est donc en train de gagner de plus en plus de points dans l'opinion des développeurs (ce qu'on appelle en anglais la *developer mindshare*). La preuve : les API de gros services (Amazon, eBay, Yahoo!, Flickr, etc.) non seulement offrent du SOAP et du REST, mais mettent souvent l'accent sur REST, voire ne proposent *que* du REST ! Dans la même veine, le framework d'applications web Ruby on Rails, réputé pour son agilité, intègre REST au cœur de son architecture à partir de sa version 1.2, de façon presque transparente (« Pour toute application développée, une API offerte ! »).

Nous utiliserons donc, dans ce chapitre, exclusivement des API REST.

Cherchons des livres sur Amazon.fr

Pour notre premier cas concret, prenons une API particulièrement connue : celle du *Web Assosicates Service* d'Amazon. L'API est la même quel que soit le site utilisé (US, UK, France, Allemagne, etc.), mais certaines opérations ne sont pas disponibles partout. Nous utiliserons l'opération la plus courante : la recherche d'éléments. Pour être plus précis, nous allons nous spécialiser sur la recherche de livres.

À l'instar de nombreux fournisseurs de services, Amazon exige un enregistrement préalable pour pouvoir utiliser son API : cet enregistrement va vous fournir une **clef API**, que vous devrez indiquer à chaque requête au service.

Certaines API Amazon sont à utilisation payante, mais celle dont nous avons besoin ici est gratuite, dans la limite toutefois d'une requête par seconde (ce qui est largement suffisant pour les besoins de cet exemple !).

Commencez par vous créer un répertoire de travail, par exemple amazon.

Obtenir une clef pour utiliser l'API

Le portail des API Amazon présente l'ensemble des API disponibles, leurs mises à jour, etc. Il est sur <http://www.amazon.com/gp/browse.html?node=3435361>. On trouve également toutes les informations techniques, pour les développeurs, sur <http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=5>.

- 1 Sur la page portail, suivez le lien central **Create** (étape 1). Vous avez alors le choix entre associer votre compte Amazon Web Services avec un compte client existant, ou créer un compte dédié.
- 2 Supposons le second cas : saisissez une adresse de courriel valide pour vous, et cochez l'option **No, I am a new customer**. Puis actionnez le bouton **Continue**.
- 3 Saisissez votre nom, confirmez l'adresse de courriel et saisissez puis confirmez un mot de passe. Actionnez le bouton **Continue**.
- 4 Saisissez les informations demandées (oui, il y en a beaucoup, mais à ma connaissance, Amazon respecte bien la confidentialité...). Pour la section **State, Province or Region**, mettez par exemple F.
- 5 Prenez soigneusement connaissance des termes de la licence d'utilisation, en particulier les sections 4 et 5.1. Pour l'essentiel, en-dehors des dispositions classiques (s'abstenir d'utiliser des logos ou marques de partenaires Amazon, d'utiliser les données obtenues pour du marketing direct ou du spam, de modifier du contenu graphique fourni par Amazon, sauf pour redimensionnement...), il s'agit de se limiter à une requête par seconde et par IP, et de ne pas envoyer de fichiers supérieurs à 40 Ko...
- 6 Cochez la case indiquant votre acceptation de la licence et actionnez le bouton **Continue**.
- 7 Et voilà ! Vous allez recevoir un e-mail à l'adresse que vous avez fournie, en général en moins de deux minutes, intitulé *Welcome to Amazon Web Services*. Vous y trouverez un lien d'obtention de clef. Cliquez dessus.
- 8 Vous arrivez sur une page d'authentification. Saisissez l'adresse de courriel associée à votre compte et votre mot de passe. Actionnez le bouton **Continue**.
- 9 Vous y voilà ! Sur la droite de la deuxième section, vous voyez s'afficher vos deux identifiants personnels : la clef API (**Your Access Key ID**) et la clef secrète (**Your Secret Access Key**), qui ne s'affiche qu'après que vous avez cliqué sur le lien **Show**. Cette clef secrète est nécessaire pour « signer » certaines opérations de l'API. Nous n'en aurons pas besoin dans ce chapitre.
- 10 Copiez-collez soigneusement la clef API dans un endroit facile d'accès, comme un fichier texte dans votre répertoire de travail.

L'appel REST à Amazon.fr

Voyons à présent l'anatomie de notre interaction avec le service.

Anatomie de la requête

Notre requête utilise le verbe GET (celui des liens et des saisies manuelles dans la barre d'adresse), sur une ressource centrale pour Amazon Web Services, identifiée par l'URL suivante :

```
http://ecs.amazonaws.fr/onca/xml
```

Nous devons fournir deux paramètres fondamentaux :

- 1 L'API utilisée, avec le paramètre `Service`. Dans notre cas, on indiquera `AWSECommerceService`.
- 2 Notre clef API, avec le paramètre `AWSAccessKeyId`.

Par ailleurs, puisque nous travaillons avec ECS, nous devons fournir un paramètre supplémentaire :

- 3 L'opération qui nous intéresse, avec le paramètre `Operation`. Pour des recherches d'éléments à la vente, on spécifie `ItemSearch`.

Cette opération a de nombreux paramètres pour affiner les résultats, qui sont tous décrits dans la documentation de l'API, à savoir dans le cas qui nous occupe :

http://docs.amazonwebservices.com/AWSECommerceService/2006-06-26/DG/CHAP_OperationListAlphabetical.html#ItemSearch.

Sur les quelque 30 paramètres possibles (dont certains sont spécifiques à un type de produit, par exemple DVD), il y a seulement deux paramètres incontournables :

- 4 `SearchIndex`, qui précise le type de produit recherché. Dans notre cas, on utilisera `Books` ou `ForeignBooks`.
- 5 `Keywords`, qui contient la « saisie de l'utilisateur », donc les mots-clefs à utiliser pour la recherche.

Un paramètre très important est `ResponseGroup`, qui précise les ensembles d'informations qu'on veut récupérer. Il existe 24 groupes possibles, certains étant une combinaison de plusieurs autres. La valeur par défaut combine `Request` (qui nous renvoie le détail de notre requête, pour validation) et `Small` (la plus petite combinaison, qui ne fournit pas, par exemple, les prix ou les numéros ISBN, encore moins les images de couvertures).

Cela ne nous suffisant pas, nous définirons :

- 6 `ResponseGroup` à `Medium`.

Cette combinaison Medium ajoute aux valeurs par défaut (titre, auteurs et contributeurs, etc.) les groupes ItemAttributes (n° ISBN, nombre de pages, dimensions, informations de publication et de fabrication...), OfferSummary (tarifs proposés), SalesRank (position dans les ventes), EditorialReview (généralement constitué de la 4^e de couverture et d'un ou deux extraits d'articles de presse) et Images (vignettes de couverture).

Voici donc un exemple d'URL (forcément très longue) pour une recherche « RSS Atom » en livres français. La valeur APIKEY est à remplacer par votre propre clef.

```
http://ecs.amazonaws.fr/onca/xml?Service=AWSECommerceService&AWSAccessKeyId=APIKEY&Operation=ItemSearch&SearchIndex=Books&Keywords=RSS%20Atom&ResponseGroup=Medium
```

Remarquez l'encodage URL de « RSS Atom », qui donne RSS%20Atom.

Allez-y, essayez-la dans votre navigateur : vous allez récupérer un document XML. Je vous conseille d'éviter MSIE 6, qui l'affichera de manière assez brouillonne. En repliant certains noeuds du document XML résultat, on obtient l'affichage de la figure 9-1, qui permet de voir l'essentiel.

Figure 9-1

Une portion de notre document XML de résultats



The screenshot shows the Mozilla Firefox browser window with the developer tools open. The address bar displays the URL: `http://ecs.amazonaws.fr/onca/xml?Service=...`. The main content area of the browser shows a partially collapsed XML document structure. The visible part starts with the root element `<ItemSearchResponse>`, which contains an `<OperationRequest>` node, followed by an `<Items>` node. Inside `<Items>`, there is one `<Item>` node. This node contains several child elements: `<ASIN>2212120281</ASIN>`, `<DetailPageURL>`, `<SalesRank>4701</SalesRank>`, `<SmallImage>`, `<MediumImage>`, and `<LargeImage>`. The `<MediumImage>` node has its `<URL>` attribute expanded, showing the URL: `http://ecx.images-amazon.com/images/I/41N65QQPORL._SL160_.jpg`. The `<LargeImage>` node also has its `<URL>` attribute expanded, showing the URL: `http://ecx.images-amazon.com/images/I/41N65QQPORL._V200_.jpg`. There are also `<ImageSets>` and `<ItemAttributes>` nodes under the `<Item>` node. The `<ItemAttributes>` node contains information about the author (Christophe Porteneuve), binding (Broché), creators (Auteur: Christophe Porteneuve, Préface: Tristan Nitot), Dewey Decimal Number (5.276), EAN (9782212120288), ISBN (2212120281), label (Eyrolles), list price (4200), currency code (EUR), and formatted price (EUR 42,00). The `<Manufacturer>` node is also present.

```

<ItemSearchResponse>
  + <OperationRequest></OperationRequest>
  - <Items>
    + <Request></Request>
    <TotalResults>2</TotalResults>
    <TotalPages>1</TotalPages>
    - <Item>
      <ASIN>2212120281</ASIN>
      + <DetailPageURL></DetailPageURL>
      <SalesRank>4701</SalesRank>
      + <SmallImage></SmallImage>
      - <MediumImage>
        - <URL>
          http://ecx.images-amazon.com/images/I/41N65QQPORL._SL160_.jpg
        </URL>
        <Height Units="pixels">160</Height>
        <Width Units="pixels">131</Width>
      </MediumImage>
      + <LargeImage></LargeImage>
      + <ImageSets></ImageSets>
      - <ItemAttributes>
        <Author>Christophe Porteneuve</Author>
        <Binding>Broché</Binding>
        <Creator Role="Auteur">Christophe Porteneuve</Creator>
        <Creator Role="Préface">Tristan Nitot</Creator>
        <DeweyDecimalNumber>5.276</DeweyDecimalNumber>
        <EAN>9782212120288</EAN>
        <ISBN>2212120281</ISBN>
        <Label>Eyrolles</Label>
        - <ListPrice>
          <Amount>4200</Amount>
          <CurrencyCode>EUR</CurrencyCode>
          <FormattedPrice>EUR 42,00</FormattedPrice>
        </ListPrice>
        <Manufacturer>Eyrolles</Manufacturer>
      </ItemAttributes>
    </Item>
  </Items>
</ItemSearchResponse>

```

Le document XML de réponse

Le nœud racine est `ItemSearchResponse`. Un nœud fils `Items` contient le nombre total de résultats et de pages, puis un nœud fils `Item` par résultat.

Chacun de ces noeuds `Item` propose entre autres :

- l'URL Amazon.fr du livre (`DetailPageURL`) ;
- la position dans les ventes (`SalesRank`) ;
- l'URL d'une vignette de taille moyenne (`URL` dans `MediumImage`) ;
- un élément de détails, `ItemAttributes`, avec entre autres :
 - le titre (`Title`) ;
 - l'auteur ou les auteurs (`Author`) ;
 - les éventuels contributeurs, traducteurs... (`Creator`) ;
 - le numéro ISBN (`ISBN`) ;
 - le nombre de pages (`NumberOfPages`) ;
 - la reliure (`Binding`) ;
 - les dimensions (`PackageDimensions` et ses fils `Height`, `Length` et `Width`) ;
 - le prix éditeur (`ListPrice`) ;
 - l'éditeur (`Publisher`) et la date de sortie (`PublicationDate`) ;
 - le plus bas prix Amazon.fr (`LowestNewPrice` ou `LowestUsedPrice` dans `OfferSummary`).

Il nous appartient d'extraire ces informations et de les afficher agréablement. Ce sera le travail d'une feuille XSLT. Cependant, ne mettons pas la charrue avant les bœufs, et commençons par construire une page de recherche Amazon.fr !

Notre formulaire de recherche

Commençons par notre formulaire de recherche. Dans la meilleure tradition de l'amélioration progressive, garante d'accessibilité, nous allons réaliser un formulaire classique. Bon, idéalement, sa cible serait une action sur notre couche serveur qui récupérerait le contenu, le formaterait, etc. Ici pour simplifier, nous utiliserons directement l'URL de la ressource cible chez Amazon.fr.

Notre JavaScript interceptera l'envoi du formulaire pour utiliser plutôt Ajax, et réaliser une transformation côté client du XML récupéré.

Nous aurons ici une maigre couche serveur (juste un proxy de téléchargement), aussi utiliserons-nous notre structure habituelle : un répertoire `docroot` pour le contenu client, et le serveur au même niveau.

Dans le répertoire docroot, nous allons créer notre page HTML et sa feuille de style. Voici index.html.

Listing 9-1 Notre formulaire de recherche Amazon.fr

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  ↪ xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Recherche Amazon (E-Commerce Service)</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Recherche Amazon (E-Commerce Service)</h1>

<form id="searchForm" method="get"
  ↪ action="http://ecs.amazonaws.fr/onca/xml">
  <p>
    <input type="hidden" name="Service" value="AWSECommerceService" /> ❶
    <input type="hidden" name="AWSAccessKeyId" value="..." />
    <input type="hidden" name="Operation" value="ItemSearch" />
    <input type="hidden" name="ResponseGroup" value="Medium" />
    <label for="cbxIndex" accesskey="C">Catégorie</label>
    <select id="cbxIndex" name="SearchIndex" size="1" tabindex="1"> ❷
      <option value="Books">Livres en français</option>
      <option value="ForeignBooks">Livres en VO</option>
    </select>
  </p>
  <p>
    <label for="edtKeywords" accesskey="M">Mots-clés</label>
    <input type="text" id="edtKeywords" name="Keywords" tabindex="2" />
  </p>
  <p>
    <input type="submit" id="btnSearch" value="Chercher !" tabindex="3" />
  </p>
</form>

<div id="indicator" style="display: none;"></div>

<div id="results"></div> ❸

</body>
</html>
```

À partir de la ligne ①, vous voyez la liste des paramètres que nous avons détaillés plus tôt. Évidemment, vous préciserez votre propre clef pour le paramètre AWSAccessKeyId. Il existe deux paramètres manipulables ici par l'utilisateur ② : le type d'élément recherché (livre en français ou livre en V.O.), et bien entendu les mots-clefs de recherche. À l'issue de la recherche et du formatage du résultat, nous injecterons le fragment XHTML obtenu dans le conteneur d'ID results ③.

Histoire que tout ceci soit un peu plus joli, nous allons y coller une feuille CSS.

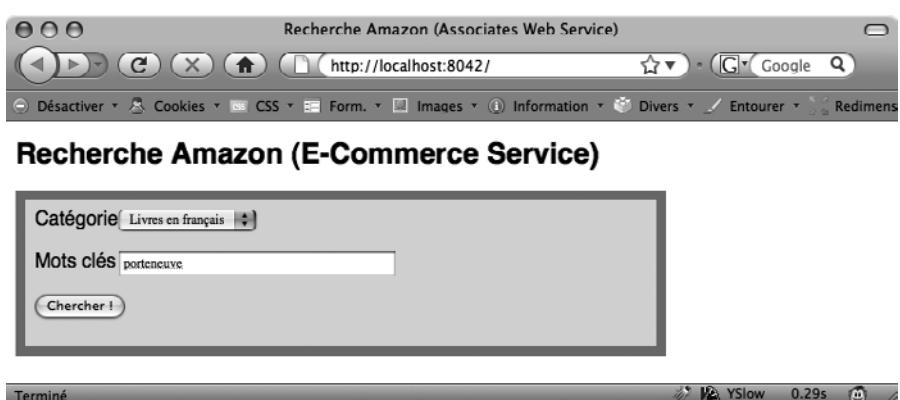
Listing 9-2 Une petite feuille de style pour y voir plus clair

```
body {  
    font-family: sans-serif;  
}  
  
h1 {  
    font-size: 150%;  
}  
  
/* FORMULAIRE */  
  
form#searchForm {  
    border: 0.5em solid #888;  
    background: #ddd;  
    width: 60ex;  
    padding: 0.5em;  
}  
  
form#searchForm p {  
    position: relative;  
    height: 2.2em;  
    margin: 0;  
}  
  
#edtKeywords, #cbxIndex {  
    font-family: serif;  
    position: absolute;  
    left: 14ex;  
}  
  
#edtKeywords {  
    width: 45ex;  
}  
  
#indicator {  
    margin-top: 1em;  
    height: 16px;  
    color: gray;
```

```
    font-size: 14px;  
    line-height: 16px;  
    background: url(spinner.gif) no-repeat;  
    padding-left: 20px;  
}
```

Ce formulaire (figure 9-2) doit déjà fonctionner tel quel : si vous saisissez une recherche et validez, vous devez obtenir un document XML Amazon.fr en réponse. Encore une fois, certains navigateurs, par exemple Firefox, affichent le XML de façon plus pratique que d'autres.

Figure 9-2
Notre formulaire
de recherche



Passer par Ajax

Interceptons maintenant l'envoi normal du formulaire pour passer plutôt par Ajax. Pour ce faire, il va toutefois nous falloir une couche serveur jouant le rôle de proxy. Et en vertu de la *Same Origin Policy*, nous devrons accéder à notre page au travers de cette même couche serveur.

La couche serveur, intermédiaire de téléchargement

Cela nous donne un serveur plutôt simple, sans rien de nouveau par rapport aux précédents. Nous avons juste factorisé le code pour qu'il détecte automatiquement la présence d'un serveur proxy entre votre poste et Internet, à l'aide de la variable Linux/Unix habituelle `http_proxy`. Sous Windows, vous aurez besoin, avant de le lancer, de définir une variable adaptée. Par exemple, pour un proxy HTTP `proxy.maboite.com`, sur le port 3128 :

```
set http_proxy=http://proxy.maboite.com:3128
```

Voici le code de `serveur.rb`, juste au-dessus de votre répertoire docroot.

Listing 9-3 Le code de notre serveur « proxy de téléchargement »

```
#!/usr/bin/env ruby

require 'net/http'
require 'uri'
require 'webrick'
include WEBrick

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, './docroot')

def get_requester
  if ENV.has_key?('http_proxy')
    comps = ENV['http_proxy'].split(':')
    host, port = comps[0..-2].join(':',), comps[-1]
    host.sub!(/https?:\/\/(.*)/, '')
    port = port.to_i rescue 80
    Net::HTTP::Proxy(host, port)
  else
    Net::HTTP
  end
end # get_requester

server.mount_proc('/xmlProxy') do |request, response| ❶
  res = get_requester.get_response(URI.parse(request.query['url'])) ❷
  if res['Content-Type'] =~ /xml/ ❸
    response['Content-Type'] = res['Content-Type']
  else
    ct = 'text/xml'
    if res.body =~ /^.+? encoding="(.*?)"\?>/
      ct += '; charset=' + $1
    end
    response['Content-Type'] = ct
  end
  response.body = res.body
end

trap('INT') { server.shutdown }

server.start
```

Notez l'URL locale pour notre fonction de proxy : `/xmlProxy` ❶, qui accepte un paramètre `url` ❷. Le bloc `if/else` en ❸ permet de forcer un contenu XML avec un jeu de caractères issu du prologue XML du document, au cas où le serveur d'origine ne renverrait pas le bon `Content-Type`.

Une fois ce serveur lancé, vous devez pouvoir accéder à votre page de recherche sur <http://localhost:8042>.

Intercepter le formulaire

C'est le moment de créer notre script client, déjà prévu dans notre HTML : `client.js`. Commencez par vérifier que vous avez bien copié `prototype.js`, depuis l'archive des codes source ou un de vos précédents répertoires de travail, dans docroot. Voici notre script.

Listing 9-4 Une première version de notre script, client.js

```
function bindForm() {
    $('searchForm').observe('submit', hookToAjax);
} // bindForm

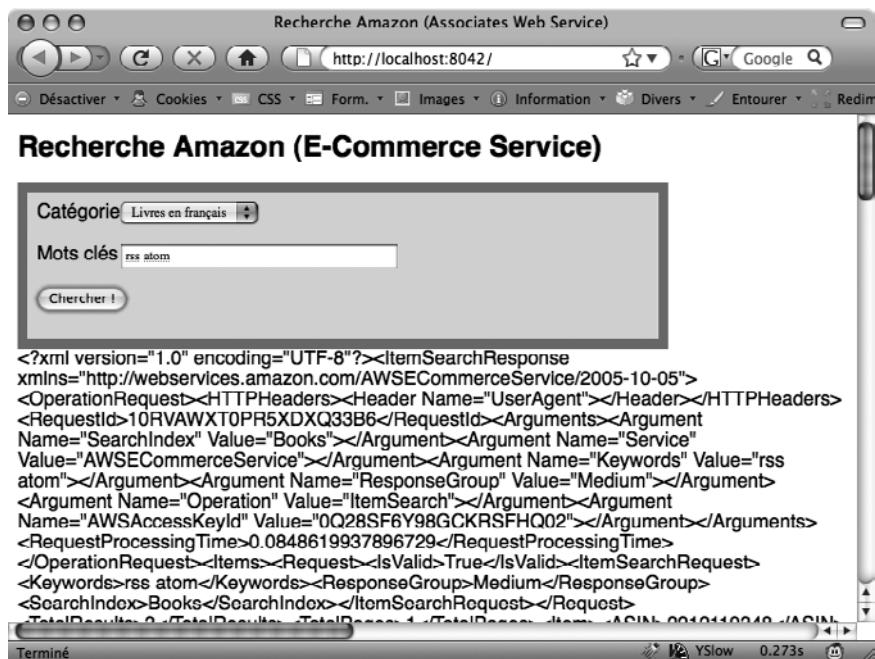
function hookToAjax(event) {
    event.stop();
    var form = $('searchForm'), indicator = $('indicator');
    indicator.update('').show(); ①
    $('results').update(''); ②
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(form.action +
            '?' + form.serialize()), ③
        onFailure: function() {
            indicator.hide();
        },
        onSuccess: function(requester) {
            indicator.update('Mise en forme'); ①
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var xml = requester.responseText; ④
                $('results').update(xml.escapeHTML()); ⑤
                indicator.hide();
            }, 10); ⑥
        }
    });
} // hookToAjax

Ajax.Responders.register({ onException: function(requester, e) {
    $('indicator').hide(); ⑦
    alert(e);
}});

document.observe('dom:loaded', bindForm);
```

- ➊ L'indicateur a ici trois états : caché, visible avec juste l'icône (requête puis réponse), et visible avec l'icône et le libellé « Mise en forme... », qui indiquera tout à l'heure la transformation XSLT. C'est pourquoi, avant de le rendre visible, on vide son contenu HTML éventuel pour redémarrer sans libellé aux recherches suivantes.
- ➋ Dans le même esprit, au cas où ce n'est pas la première recherche, on vide le conteneur de résultats de son précédent contenu.
- ➌ Remarquez les précautions d'encodage pour s'assurer que toute l'URL ne constitue bien qu'un seul paramètre. Et c'est l'utilisation rêvée d'un `Form.serialize` (vous ne savez plus ce que cela fait ? courez au chapitre 4 !).
- ➍ Notre proxy de téléchargement garantissant un type MIME XML, le navigateur produira un DOM pour le XML renvoyé, et le mettra à disposition dans la propriété `responseXML` du requêteur. Comme on ne le transforme pas pour l'instant, prenons plutôt `responseText` pour pouvoir l'afficher.

Figure 9–3
Le résultat injecté
dans le conteneur
`results`



- ⑥ Ce `setTimeout` de 10 millisecondes autour du code « lourd » (requête Ajax, traitement de la réponse) est une astuce courante pour s'assurer que le navigateur va bien prendre le temps (puisque il a 10 ms devant lui) d'afficher notre libellé tout frais dans l'indicateur. Autre option : appeler `defer()` sur la fonction anonyme.
- ⑦ C'est la première fois que nous utilisons `Ajax.Responders`, qui a été décrit en fin de chapitre 7. Ainsi, nous affichons toute exception survenue lors d'un traitement Ajax. On ne sait jamais...

Allez, on rafraîchit la page en prenant soin d'ignorer le cache éventuel, et on retente une recherche. Vous devez normalement voir apparaître le résultat (figure 9-3).

Si vous examinez attentivement le XML retourné, vous apercevez effectivement les éléments que nous avons déjà vus : `Item`, `ItemAttributes`, `Title`, etc.

De XML à XHTML : la transformation XSLT

Utilisons maintenant une feuille de styles XSLT pour transformer, presque d'un coup de baguette magique, ce magma XML en un joli fragment XHTML. Pour cela, nous avons plusieurs étapes à suivre :

- 1 Écrire la feuille XSLT.
- 2 Doter notre page de capacités XPath et XSLT.
- 3 La télécharger côté client, au chargement de la page.
- 4 Changer le traitement du résultat XML : au lieu d'afficher le XML, on le transforme et on affiche le résultat final.
- 5 Augmenter la CSS et ajouter quelques astuces JavaScript pour embellir le XHTML obtenu.

Notre feuille XSLT

Créez un sous-répertoire `xsl` dans `docroot`, et mettez-y le fichier `books.xsl` suivant (ou reprenez-le de l'archive des codes source du livre, parce que c'est un beau pavé)

Listing 9-5 Notre feuille XSLT

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
    <xsl:apply-templates select="//Items"/> ①
</xsl:template>
```

```
<xsl:template match="Items">
    <xsl:variable name="viewedItems" select="count(Item)"/>
    <xsl:variable name="totalItems" select="TotalResults"/>
    <xsl:choose>
        <xsl:when test="$viewedItems > 0">
            <p class="resultCount">
                <xsl:value-of select="$totalItems"/> résultat(s) ②
                <xsl:if test="$viewedItems < $totalItems">
                    dont <xsl:value-of select="$viewedItems"/>
                    listés ici.
                </xsl:if>
            </p>
            <dl>
                <xsl:apply-templates select="Item"/> ①
            </dl>
        </xsl:when>
        <xsl:otherwise>
            <p class="resultCount">Aucun résultat !</p>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
<xsl:template match="Item"> ③
    <dt>
        <xsl:element name="a"> ④
            <xsl:attribute name="href">
                <xsl:value-of select="DetailPageURL"/>
            </xsl:attribute>
            <xsl:value-of select="ItemAttributes/Title"/>
        </xsl:element>
    </dt>
    <dd>
        <xsl:element name="img">
            <xsl:attribute name="class">cover</xsl:attribute>
            <xsl:attribute name="src">
                <xsl:value-of select="MediumImage/URL"/>
            </xsl:attribute>
        </xsl:element>
        <ul>
            <li>
                Auteur(s) :
                <xsl:for-each select="ItemAttributes/Author
                    ↪ |ItemAttributes/Creator">
                    <xsl:sort select="@Role"/>
                    <xsl:element name="span">
                        <xsl:attribute name="class">
                            <xsl:choose>
                                <xsl:when test="not(@Role)">author
                                </xsl:when>
                                <xsl:otherwise>contributor</xsl:otherwise>
                            </xsl:choose>
                        </xsl:attribute>
                    </xsl:element>
                </xsl:for-each>
            </li>
        </ul>
    </dd>
</xsl:template>
```

```
</xsl:attribute>
<xsl:value-of select=". "/>
<xsl:if test="@Role"> (<xsl:value-of select="@Role"/>)
</xsl:if>
<xsl:if test="position() != last()">, </xsl:if>
</xsl:element>
</xsl:for-each>
</li>
<li>
<xsl:value-of select="ItemAttributes/Binding"/>,
<xsl:value-of select="ItemAttributes/NumberOfPages"/> pages,
<xsl:value-of
    ↪ select="ItemAttributes/PackageDimensions/Height"/> &times;
<xsl:value-of
    ↪ select="ItemAttributes/PackageDimensions/Length"/> &times;
<xsl:value-of
    ↪ select="ItemAttributes/PackageDimensions/Width"/>
</li>
<li>
    Publi&#233; par <xsl:value-of
        ↪ select="ItemAttributes/Publisher"/> le
    <span class="date"><xsl:value-of
        ↪ select="ItemAttributes/PublicationDate"/></span>
</li>
<li>ISBN : <xsl:value-of select="ItemAttributes/ISBN"/></li>
<li>
    Prix &#233;diteur : <span class="price"><xsl:value-of
        ↪ select="ItemAttributes/ListPrice/Amount"/></span>.
    Prix Amazon.fr :
    <xsl:choose> ⑤
        <xsl:when test="OfferSummary/LowestNewPrice">
            <span class="price"><xsl:value-of
                ↪ select="OfferSummary/LowestNewPrice/Amount"/></span>.
        </xsl:when>
        <xsl:when test="OfferSummary/LowestUsedPrice">
            <span class="price"><xsl:value-of
                ↪ select="OfferSummary/LowestUsedPrice/Amount"/></span>.
        </xsl:when>
        <xsl:otherwise>N/D</xsl:otherwise>
    </xsl:choose>
</li>
<li>
    <xsl:variable name="availableCount" select="OfferSummary/TotalNew"/>
    <xsl:variable name="usedCount" select="OfferSummary/TotalUsed"/>
    <xsl:choose> ⑥
        <xsl:when test="$availableCount > 5">En stock</xsl:when>
        <xsl:when test="$availableCount > 0">
            <xsl:value-of select="$availableCount"/> exemplaire(s) neuf(s)
        </xsl:when>
```

```
<xsl:when test="$usedCount > 0">
    <xsl:value-of select="$usedCount"/> exemplaire(s) d'occasion
</xsl:when>
<xsl:otherwise>Ce livre n'est pas en stock actuellement.
    ↪ <xsl:otherwise>
</xsl:choose>
</li>
<li>
    Position dans les ventes :
    <xsl:value-of select="SalesRank"/>
</li>
</ul>
</dd>
</xsl:template>
</xsl:stylesheet>
```

Eh bien, quel monstre ! Nous avons en effet souhaité faire un exemple un peu constant... Si vous n'avez jamais fait de XSLT, vous aurez peut-être du mal à tout suivre. Nous vous recommandons vivement d'aller jeter un œil aux ressources en ligne citées en fin de chapitre.

Voici quelques précisions :

- ➊ Ces `xsl:apply-templates` appliquent des modèles (*templates*) à une série de noeuds XML extraits à l'aide de l'attribut `select`.
- ➋ L'entité numérique #233 représente le « é », qu'il vaut mieux ne pas laisser littéral, car certains navigateurs n'utilisent pas correctement la déclaration de jeu de caractères dans le prologue XML de notre feuille XSLT, et croient qu'il s'agit d'UTF-8. L'entité numérique sera interprétée correctement.
- ➌ C'est ce `template` qui contient toute la description de transformation pour un élément de résultat (pour un livre, donc).
- ➍ On utilise `xsl:element` quand on veut générer dynamiquement les valeurs de certains attributs (par exemple `href` ou `src`) pour l'élément. Sinon, on peut utiliser l'élément directement, comme c'est le cas pour de nombreux `p`, `dl`, `dt`, `dd`...
- ➎ Un `xsl:choose` exprime une alternative, comme un `switch` en C/C++/Java/C# ou un `case...of` en Delphi. On utilise un ou plusieurs `xsl:when` (le premier qui correspond gagne, on ignore les suivants), et éventuellement un `xsl:otherwise` pour les cas restants. Ici, on gère les différents cas pour les prix : prix du neuf s'il existe, sinon prix de l'occasion s'il existe, sinon N/D (cas très rare).
- ➏ Même mécanisme, cette fois-ci pour le nombre d'exemplaires. Nous avons également décidé (arbitrairement, mais cela ressemble au comportement officiel du site) qu'à partir de 6 exemplaires neufs, on dit juste « en stock », tandis qu'en dessous on précise (ce qui incite à l'achat en cas d'hésitation).

Apprendre XSLT à notre page

Sous le répertoire docroot, créez un répertoire ajaxslt avec les fichiers que nous avions utilisés au chapitre 6 : misc.js, dom.js, xpath.js. Ajoutez-en un nouveau, que vous trouverez dans l'archive originale : xslt.js. Je vous rappelle qu'il vous faut au minimum Google AJAXSLT 0.5 (vous avez des versions suffisantes dans l'archive des codes source).

Il nous reste bien sûr à charger ces scripts, en modifiant l'en-tête de notre page HTML comme ceci.

Listing 9-6 Le début modifié de notre index.html, qui charge AJAXSLT

```
<script type="text/javascript" src="prototype.js"></script>
<script type="text/javascript" src="ajaxslt/misc.js"></script>
<script type="text/javascript" src="ajaxslt/dom.js"></script>
<script type="text/javascript" src="ajaxslt/xpath.js"></script>
<script type="text/javascript" src="ajaxslt/xslt.js"></script>
<script type="text/javascript" src="client.js"></script>
```

Et voilà, cela suffit, notre page parle XPath et XSLT ! Si seulement cela fonctionnait aussi simplement pour tous les lecteurs...

Charger la feuille XSLT au démarrage

C'est là que nous allons commencer à faire des choses rigolotes... D'abord, il faut charger la feuille XSLT au démarrage, et en stocker le DOM pour utilisation ultérieure (à chaque résultat de recherche). On utilisera donc... une variable globale (d'accord, ça, ce n'est pas rigolo).

En revanche, tant que ce DOM n'est pas disponible, lancer une recherche est périlleux : si celle-ci, par extraordinaire, aboutit avant notre chargement XSLT, la page sera incapable de réaliser la transformation. Nous allons donc désactiver initialement le bouton d'envoi du formulaire, et ne le réactiver qu'une fois la feuille XSLT chargée !

Commençons par modifier notre script. Ajoutons tout en haut notre déclaration de variable :

```
var xsltSheet;
```

La fonction bindForm est déjà appelée au démarrage de la page. Même si son nom n'est du coup plus très approprié (`initPage` lui irait mieux, mais `bindForm`, c'est déjà mieux que `gloubiBoulga`), nous allons lui ajouter une requête Ajax sur notre document XSLT. Une fois la feuille récupérée, le bouton d'envoi du formulaire est activé.

Listing 9-7 Ajout du chargement XSLT à notre initialisation

```
function bindForm() {
    new Ajax.Request('xsl/books.xsl', {
        method: 'get',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            $('#btnSearch').disabled = false;
        }
    });
    $('searchForm').observe('submit', hookToAjax);
} // bindForm
```

Les lecteurs observateurs et futés (mais si, vous aussi !) poseront sans doute la question : mais pourquoi donc analyser manuellement `responseText` avec `xmlParse`, plutôt que d'utiliser directement `responseXML` ? Tout simplement parce que pour utiliser `responseXML`, il faut avoir la garantie que le serveur a renvoyé un type MIME XML pour notre fichier `books.xsl`. Ce n'est pas toujours le cas (cela dépend de sa configuration), et dans notre mini-serveur WEBrick, ce n'est *pas* le cas. Donc on se blinde, et on analyse la feuille manuellement.

Dernière touche : il faut désactiver par défaut le bouton d'envoi. Modifions donc sa définition dans `index.html`, comme ceci :

```
<input type="submit" id="btnSearch" value="Chercher !"
    ↪ tabindex="3" disabled="disabled" />
```

Effectuer la transformation

Allez, on y est presque ! Maintenant que nous avons une feuille XSLT disponible, et la capacité à l'utiliser pour une transformation, changeons notre traitement des résultats de recherche. Modifions donc la fonction `onSuccess` de notre requête Ajax, dans `client.js`.

Listing 9-8 Notre traitement de résultat XML, mis à jour

```
onSuccess: function(requester) {
    indicator.update('Mise en forme…');
    var tmr = window.setTimeout(function() {
        window.clearTimeout(tmr);
        var html = xsltProcess(requester.responseXML, xsltSheet);
        $('#results').update(html);
        indicator.hide();
    }, 10);
}
```

Il nous reste un minuscule détail à régler : par défaut, Google AJAXSLT affiche un journal de ses principales étapes de traitement, dans un pavé en haut à droite de la page. Pour éviter cela, ajoutons en fin de script, juste avant les inscriptions d'observateurs, la ligne suivante :

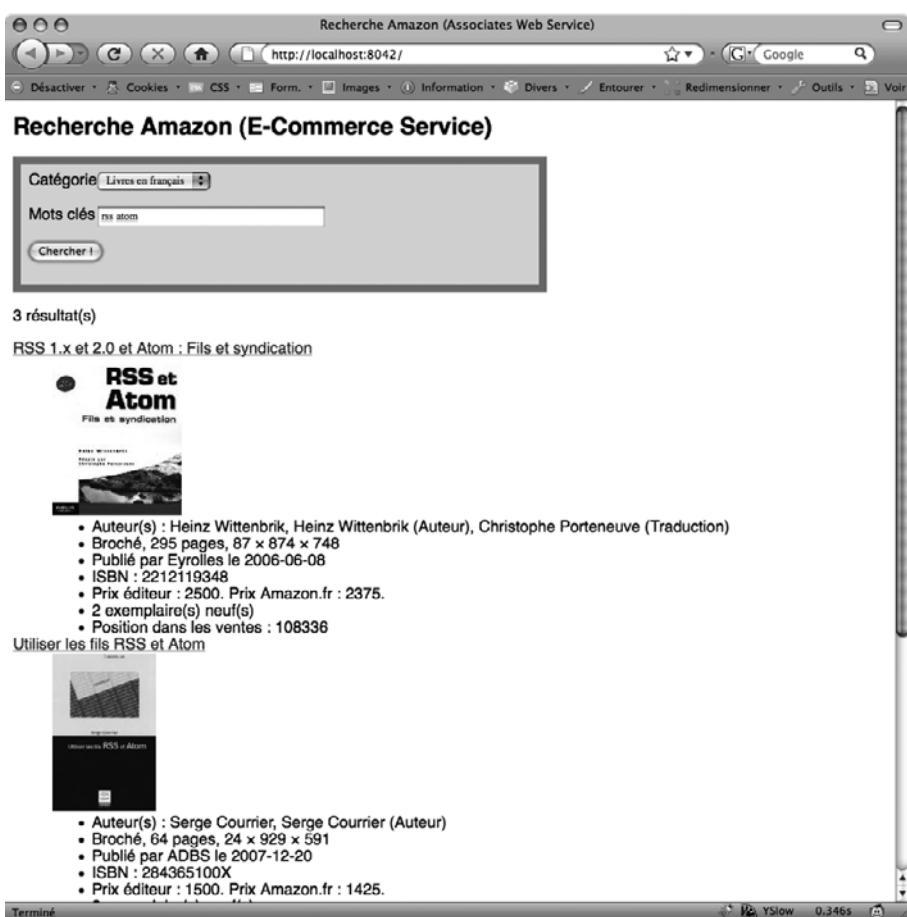
```
logging__ = false;
```

Cette variable globale est définie par Google AJAXSLT, et vaut `true` par défaut. Passée à `false`, elle réduit les moteurs de traitement au silence.

Vous voyez ? Ce n'était pas si difficile (d'ailleurs, si la simplicité vous frustre, je vous suggère de recoder `xsltProcess` ou d'oublier ce livre). Observons le résultat, après un rechargement strict de la page et une nouvelle requête (figure 9-4).

Figure 9-4

Nos résultats transformés !



Franchement, ce n'est pas si mal ! Mais on va améliorer tout ça...

Embellir le résultat

Commençons par ajouter quelques jolies choses à notre feuille CSS.

Listing 9-9 Ajouts CSS destinés aux résultats de recherche

```
/* RÉSULTATS DE LA RECHERCHE */

span.contributor {
    color: #666;
    font-size: 90%;
}

#results dt {
    font-weight: bold;
    font-size: 120%;
}

#results dd {
    margin: 0.5em 0 0.5em 1em;
    width: 50em;
    height: 160px;
    position: relative;
    padding: 0.25em;
    border: 1px solid gray;
}

#results dd ul {
    position: absolute;
    left: 150px;
    top: 5px;
}
```

Voyons ce que cela donne (figure 9-5)...

Il faut peu de chose... Ceci dit, peut-être avez-vous remarqué deux fausses notes dans ce joli résultat :

- 1 Les dates utilisent un format plutôt technique : 2006-06-08 par exemple. Ce n'est pas terrible...
- 2 Les prix sont prohibitifs (à moins qu'il ne s'agisse de dollars zimbabwéens ou de livres turques) : 2 375 le livre RSS, on le sent passer ! En réalité, Amazon.fr, comme tout site de commerce électronique, stocke les prix en unités basses (ici les cents, ou centimes d'euro) pour éviter toute erreur d'arrondi.

Figure 9–5

Avec un peu de CSS,
tout va mieux...

The screenshot shows a web browser window with the title "Recherche Amazon (E-Commerce Service)". The address bar indicates the page is at <http://localhost:8042/>. The search form has a "Catégorie" dropdown set to "Livres en français" and a "Mots clés" input field containing "rss atom". A "Chercher !" button is below the input fields. Below the form, a message says "3 résultat(s)". The main content area is titled "RSS 1.x et 2.0 et Atom : Fils et syndication" and displays a book cover for "RSS et Atom" by Heinz Wittenbrik. To the right of the cover is a list of details:

- Auteur(s) : Heinz Wittenbrik, Heinz Wittenbrik (Auteur), Christophe Porteneuve (Traduction)
- Broché, 295 pages, 87 x 874 x 748
- Publié par Eyrolles le 2006-06-08
- ISBN : 2212119348
- Prix éditeur : 2500. Prix Amazon.fr : 2375.
- 2 exemplaire(s) neuf(s)
- Position dans les ventes : 108336

Below this section is another titled "Utiliser les fils RSS et Atom" featuring a different book cover. To its right is a list of details:

- Auteur(s) : Serge Courier, Serge Courier (Auteur)
- Broché, 64 pages, 24 x 929 x 591
- Publié par ADBS le 2007-12-20
- ISBN : 2843865100X
- Prix éditeur : 1500. Prix Amazon.fr : 1425.
- 2 exemplaire(s) neuf(s)
- Position dans les ventes : 120029

The final section is titled "RSS et Atom : Créez vos flux de syndication et vos contenus audio et vidéo" and shows a third book cover. To its right is a list of details:

- Auteur(s) : Michel Martin, Michel Martin (Auteur)
- Broché, 234 pages, 83 x 819 x 575
- Publié par Campus Press le 2006-04-13
- ISBN : 2744020613
- Prix éditeur : 1500. Prix Amazon.fr : 1300.
- 2 exemplaire(s) neuf(s)
- Position dans les ventes : 266547

3 Et puis il y a la question des dimensions... Depuis la version 4.0 de l'API, Amazon a uniformisé ses tailles et travaille en centièmes de pouces ! Ça nous fait tout de même 0,0254 cm, alors il va falloir multiplier tout ça à l'aide d'une petite formule...

Nous ne pouvons pas laisser la page comme cela. Il faut trouver une astuce. Pour la conversion des centièmes de pouces vers les centimètres, on peut tirer parti des nombreuses fonctions XPath de découpe de texte et de calcul. On remplacera simplement le bloc de dimensions par ce qui suit :

```
<xsl:value-of select="translate(round(number(ItemAttributes/
PackageDimensions/Height)*2.54) div 100, '.', ',')"/> &times;
<xsl:value-of select="translate(round(number(ItemAttributes/
PackageDimensions/Length)*2.54) div 100, '.', ',')"/> &times;
<xsl:value-of select="translate(round(number(ItemAttributes/
PackageDimensions/Width)*2.54) div 100, '.', ',')"/> (cm)
```

D'accord, tout ceci n'est pas très factorisé, mais pour éviter la redondance il faudrait soit créer un template dédié et l'appeler en restreignant le contexte (ce qui serait encore plus verbeux qu'ici), soit sortir d'XSLT et passer la main à JavaScript comme nous allons le faire pour le reste. Mais cela supposerait de baliser chaque taille pour traitement ultérieur, ce qui ne nous avancerait pas beaucoup non plus. Autant laisser comme ça.

Ce type d'astuce ne permet tout de même pas de gérer complètement la question de la date, même si c'est suffisant pour s'occuper des prix. On va donc recourir à un traitement JavaScript sur le XHTML obtenu, par nécessité pour la date et par souci d'alternative pour les prix. C'est d'ailleurs pour ça que notre XSLT a pris soin d'isoler les prix dans des `...`, et la date de publication dans un `...`.

On va créer une fonction `adjustData`, qui prend le XHTML original et renvoie le XHTML modifié. Les dates seront affichées joliment, du style « 4 avril 2005 » ou « 1^{er} juin 2006 ». Les tarifs auront deux décimales et le symbole € en suffixe. Ajoutons tout cela au début de notre script `client.js`. Commençons par déclarer les noms de mois, le format des prix et les expressions rationnelles d'extraction des textes à modifier.

Listing 9-10 Nos constantes de traitement des textes à localiser

```
MONTH_NAMES = $w('janvier février mars avril mai juin juillet août  
septembre octobre novembre décembre');  
PRICER = new Template('#{euros},#{cents}&euro;');  
RE_DATE = '(<span class="date">)([0-9-]+)(</span>)';  
RE_PRICE = '(<span class="price">)([0-9]+)(</span>)';
```

Avant de voir le code de notre fonction `adjustData`, il est bon de rappeler quelques notions.

D'abord, nous allons utiliser la méthode étendue `gsub`, transmise aux `String` par Prototype. Cette méthode accepte comme deuxième argument une fonction pour produire le texte de remplacement à chaque correspondance de texte. Cette fonction récupère un objet représentant la correspondance, lequel objet est une sorte de tableau des groupes isolés par l'expression rationnelle. Par exemple, dans l'expression rationnelle suivante :

```
(<span class="date">)([0-9-]+)(</span>)
```

nous avons trois groupes, délimités par les parenthèses : le groupe 1, qui contiendra fatallement ``, le groupe 2, qui contiendra le texte de notre date, et le groupe 3, qui contiendra nécessairement ``. Le texte que nous renvoyons

doit préserver les groupes 1 et 3, afin de permettre, par exemple, une manipulation CSS ou JavaScript. On ne change que le groupe 2.

Pour convertir un texte aaaa-mm-jj dans le format que nous souhaitons, nous allons commencer par le transformer en un tableau de nombres [a, m, j], que nous utiliserons pour obtenir le texte final.

Quant aux prix, il s'agit de retenir les deux chiffres de droite comme cents, et de garder ceux plus à gauche comme euros. Il suffit de synthétiser un objet avec deux propriétés euros et cents, contenant ces fragments textuels, pour pouvoir utiliser l'objet Prototype Template déclaré dans PRICER.

Ces points étant éclaircis, voici le code de notre fonction `adjustData`.

Listing 9-11 Notre fonction `adjustData`, qui retouche le XHTML produit

```
function adjustData(text) {
    text = text.gsub(RE_DATE, function(match) {
        var comps = match[2].split '-';
        comps = comps.map(function(s) { return parseInt(s, 10); });
        if (1 == comps[2])
            comps[2] = '1er';
        var date = comps[2] + ' ' + MONTH_NAMES[comps[1] - 1] +
                   ' ' + comps[0];
        return match[1] + date + match[3];
    });
    text = text.gsub(RE_PRICE, function(match) {
        var centsPos = match[2].length - 2;
        var price = {
            euros: match[2].substring(0, centsPos),
            cents: match[2].substring(centsPos)
        };
        return match[1] + PRICER.evaluate(price) + match[3];
    });
    return text;
} // adjustData
```

Déclarer la fonction ne suffit pas : encore faut-il l'utiliser. Insérons donc un appel dans notre fonction de rappel `onSuccess` pour la recherche.

Listing 9-12 Notre fonction `onSuccess` mise à jour

```
onSuccess: function(requester) {
    indicator.update('Mise en forme');
    var tmr = window.setTimeout(function() {
        window.clearTimeout(tmr);
        var html = xsltProcess(requester.responseXML, xsltSheet);
```

```

        html = adjustData(html);
        $('results').update(html);
        indicator.hide();
    }, 10);
}

```

Un rafraîchissement strict et une nouvelle recherche plus tard, nous obtenons quelque chose de similaire à la figure 9-6.

Figure 9-6

Tout est dans les détails...

Cette fois, ça y est ! Nous avons terminé. Alors, comment trouvez-vous cela ? C'est sympathique, non ? Précisons au passage que, si vous souhaitez faire effectuer le traitement XSLT par Amazon directement (par exemple, si vous avez une énorme XSLT et d'énormes grappes de résultats, et si vos postes clients sont des charrues préhistoriques), c'est possible, à l'aide du paramètre `Style`, qui précise une URL, accessible sur Internet, pour votre feuille XSLT. Pour ce chapitre toutefois, l'intérêt est de réaliser le traitement totalement sur le client (ce qui décharge votre couche serveur).

Bon, je vous laisse jouer avec une bonne heure et le montrer avec exubérance aux collègues. C'est votre heure de gloire, vous l'avez bien méritée (surtout si vous avez tapé le XSLT à la main !).

Quand vous aurez fini, rejoignez-nous à la prochaine section.

Rhumatismes 2.0 : prévisions météo

Nous allons reprendre le même mécanisme général, cette fois-ci sur un autre service très populaire : les prévisions météo. Nous utiliserons le service de The Weather Channel® (« TWC »), qui est plutôt fiable, détaillé, et a en outre l'avantage de proposer une couverture mondiale.

Théoriquement, l'utilisation de ce service est soumise à l'enregistrement d'une clef API, à fournir dans chaque requête. On s'enregistre à partir de la page suivante : <https://registration.weather.com/ursa/xmloap/step1>. Toutefois, en ajustant un tout petit peu l'URL utilisée, notamment en évitant de dire qu'on utilise le service dans le cadre de sa variante « clef », on obtient les mêmes données, plus vite, et sans clef !

Si ce type de comportement est pratique pour les tests, et peut vous éviter un enregistrement fastidieux, il n'est toutefois pas forcément licite, et je ne garantis aucunement qu'il fonctionnera indéfiniment.

Si vous devez utiliser ce service en production, prenez le temps de vous enregistrer, et respectez scrupuleusement les conditions d'utilisation qui, soit dit en passant, sont un peu contraignantes (par exemple, vous devez vous limiter à un lieu à la fois, indiquer que les données sont fournies par TWC, fournir un lien sur la page d'accueil de leur site, plus trois liens promotionnels ailleurs, être un service gratuit...).

Nous allons réaliser une page permettant d'une part de saisir une ville (le nom une fois complet, nous listerons les variantes possibles, par exemple « Paris, France » et « Paris, Illinois »), et d'autre part d'afficher les prévisions à quatre jours pour cette ville, une fois celle-ci validée. Par défaut, la page utilisera Paris, en France.

La figure 9-7 montre l'aspect que doit avoir la page.

C'est tout mignon, non ? Le jeu d'icônes de grande taille (160×160 pixels) utilisé, qui comprend près de 50 imagettes correspondant aux très nombreux codes (parfois redondants) de situation fournis par TWC, est quant à lui libre de droits : il s'agit d'un jeu d'icônes gratuit de Stardock. Il est disponible dans l'archive des codes source de ce livre. Vous trouverez des alternatives sur <http://www.samurize.com/modules/ipboard/index.php?showtopic=3857>. Elles sont toutes plus jolies que le jeu fourni par défaut dans le SDK de TWC...

Figure 9–7
Notre page une fois terminée



Notez que l'heure affichée dans la partie haute est l'heure **locale** au lieu indiqué. Afin d'illustrer quelques petites finesse, nous ferons en sorte qu'elle soit mise à jour automatiquement par la suite (toutes les secondes), comme une horloge. La page recharge automatiquement les prévisions toutes les 2 heures, ce qui correspond à l'intervalle de mise à jour des prévisions à 4 jours comme indiqué dans le SDK de TWC.

Préparer le projet

Commencez par créer un répertoire de travail `meteo`. Recréez-y les incontournables : notre `serveur.rb` (strictement le même que pour notre exemple `Amazon.fr`), le répertoire `docroot`, ses fichiers `prototype.js` et `spinner.gif`, et son sous-répertoire `ajaxslt` avec son contenu.

Nous utiliserons bien entendu des fichiers `client.js` et `client.css`, que nous réécrirons complètement en revanche. Vous pouvez partir du `index.html` de notre exemple `Amazon.fr` pour réaliser celui de cet exemple. Dans la même série, on pourra reprendre le squelette externe de `xsl/books.xsl` pour nos *deux* feuilles XSLT dans cet exemple : `search.xsl` et `forecast.xsl`.

Commençons par poser la page HTML et son style de base. J'ai mis en exergue les changements par rapport à `Amazon.fr`.

Listing 9-13 Une première version de notre page index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
      xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=iso-8859-15" />
  <title>Fébô&nbsp;? Fépabô&nbsp;?</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="ajaxslt/misc.js"></script>
  <script type="text/javascript" src="ajaxslt/dom.js"></script>
  <script type="text/javascript" src="ajaxslt/xpath.js"></script>
  <script type="text/javascript" src="ajaxslt/xslt.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>

<h1>Fébô&nbsp;? Fépabô&nbsp;?</h1>

<div id="results"></div>

<div id="indicator" style="display: none;"></div>

</body>
</html>
```

Notre CSS est pour l'instant très basique.

Listing 9-14 Une première CSS basique, client.css

```
* {
  font-size: 1em;
}

body {
  font-family: sans-serif;
}

h1 {
  font-size: 150%;
}

#indicator {
  position: absolute;
  left: 1em;
```

```
    bottom: 1em;
    height: 16px;
    color: gray;
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}
```

Notez que, cette fois-ci, nous positionnons l'indicateur de progression en bas à gauche de la page. C'est un parti pris simple, qui nous permet de toujours avoir l'indicateur au même endroit alors que le reste de la page va varier en taille, ce qui évitera par exemple un décalage du bloc de prévision lorsqu'on recherche des correspondances de lieu...

Nous voici avec une page qui ne fait rien et le fait bien. Il est temps de lui donner un peu de nerf...

Récupérer les prévisions d'un lieu

Nous allons pour l'instant définir le code du lieu « en dur » dans le script. Il ne tiendra qu'à nous de le faire varier par la suite, à l'aide d'une sélection de lieu auprès de TWC.

Requête et réponse REST

L'API REST de TWC est très simple. Nous allons utiliser une première ressource, qui correspond au lieu voulu. Ce lieu est identifié par un code spécifique. Pour Paris en France, ce code est FRXX0076. L'URL de la ressource est donc <http://xoap.weather.com/weather/local/FRXX0076>. Nous y ajoutons un paramètre `dayf` (*days of forecast*) qui indique le nombre de jours de prévisions, et un paramètre `unit`, très utile, pour récupérer des valeurs en système métrique (°C, vitesses en m/s ou km/s, etc.). Au final, notre URL a l'aspect suivant :

```
| http://xoap.weather.com/weather/local/FRXX0076?dayf=4&unit=m
```

La grappe XML renvoyée n'utilise pas un type XML, ce qui est une faute de configuration de TWC. Il nous faudra l'analyser manuellement avec `responseText` et `xmlParse`, comme pour nos feuilles XSLT. Elle a l'aspect suivant.

Listing 9-15 Vue partielle du résultat XML de demande de prévisions

```

<weather ver="2.0">
  ...
  <loc id="FRXX0076">
    <dnam>Paris, France</dnam> ①
    <tm>4:57 PM</tm> ②
  ...
  </loc>
  <dayf>
    <lsup>9/5/06 3:21 PM Local Time</lsup> ③
    <day d="0" t="Tuesday" dt="Sep 5"> ④
      <hi>N/A</hi> ⑤
      <low>16</low>
    ...
    <part p="d"> ⑥
      <icon>44</icon> ⑦
      <t>N/A</t> ⑧
    ...
    </part>
    <part p="n"> ⑥
      <icon>33</icon>
      <t>Mostly Clear</t> ⑧
    ...
    </part>
  </dayf>
</weather>
```

Ce n'est pas une grappe compliquée. Voici quelques précisions pour bien comprendre son fonctionnement :

- ① `weather/loc/dnam` donne le nom complet du lieu.
- ② `weather/loc/tm` donne l'heure locale du lieu. Afin de pouvoir manipuler cette heure par la suite (notre fameuse horloge), nous devrons interpréter correctement cette chaîne de caractères pour en faire un objet JavaScript `Date`.
- ③ `weather/dayf/lsup` donne le moment de dernière mise à jour des données de mesure, en heure locale du lieu. Nous devrons là aussi extraire cette information

pour en faire un objet Date, afin de pouvoir le reformater selon la langue active du navigateur, ce qui sera plus agréable pour l'utilisateur.

- ④ Chaque jour fait l'objet d'un élément day dans weather/dayf. L'attribut d est un numéro d'ordre (0 est aujourd'hui, etc.). Bien que dans la pratique, cela semble être toujours dans l'ordre, rien ne le garantit dans la documentation : nous trierons donc dans la XSLT, à tout hasard. Notez aussi l'attribut t, qui contient le nom anglais du jour de la semaine. Ce serait pénible à obtenir autrement, alors on utilisera une astuce JavaScript pour transformer en noms français.
- ⑤ Dans un day, les éléments hi et low donnent les températures maximale et minimale. L'unité est ici le °C, puisque nous avons demandé un résultat en système métrique. Notez que parfois, on n'a pas l'information complète (il arrive même que les *deux* soient N/A). Il faudra le gérer dans la XSLT.
- ⑥ Chaque day a deux parties de détail, sous forme d'éléments part. Leur attribut p indique la période ('d' pour le jour, 'n' pour la nuit). Par souci de simplicité, nous n'utiliserons que les parties 'd'. Il est vrai que nous pourrions utiliser un paramètre XSLT pour indiquer si on souhaite, pour le premier jour (aujourd'hui), avoir l'information diurne ou nocturne, en fonction de l'heure locale du lieu... Nous vous laissons jouer avec cela !
- ⑦ Dans une part, l'élément icon indique la représentation graphique à utiliser pour les conditions météo. La valeur va de 0 à 47. Vous trouverez dans l'archive des codes source, dans le répertoire de cet exemple, un répertoire icons avec tous les fichiers PNG correspondants, que vous pouvez reprendre et placer dans votre docroot si vous réalisez l'exemple vous-même.
- ⑧ Enfin, l'élément t d'une part fournit un libellé anglais pour les conditions météo. Pour le coup, on s'épargnera une table de traductions... Nous nous contenterons de la V.O., que nous placerons dans les attributs alt et title de l'image, à condition bien sûr de ne pas tomber sur N/A.

On peut être surpris par les noms pour le moins abscons des éléments et attributs : dnam, tm, d, t, p... Ceci dit, outre le fait qu'ils sont à peu près cohérents (respectivement *data name*, *time*, *day*, *title*, *part*), ils sont nécessaires pour un service utilisé plusieurs centaines de milliers de fois par seconde au niveau mondial.

XML est un format textuel verbeux, lourd en bande passante ; aussi allons-nous tenter de réduire la taille des données. Évidemment, pour réduire vraiment la taille, il aurait été bien plus malin de retirer tout formatage de la grappe (retours chariot, indentation)...

Initialisation de la page et obtention des prévisions

Nous allons donc créer notre script de façon à ce qu'il récupère les prévisions au démarrage. Nous stockerons la feuille XSLT dans une variable globale, bien entendu. Nous stockerons également les informations de prévision (code de lieu, nombre de jours) dans un objet sur lequel on évaluera un Template représentant l'URL à requêter.

Voici le début de notre script client.js.

Listing 9-16 Le début de notre script client.js

```
DEFAULT_LOCATION = 'FRXX0076';
FORECAST_DAYS = 4;
FORECAST_REFRESH = 2 * 60 * 60 * 1000;
FORECAST_TEMPLATE = new Template(
    'http://xoap.weather.com/weather/local/#{{code}}?dayf=#{{daysAhead}}&unit=m');

var gForecastParams = {
    code: DEFAULT_LOCATION,
    daysAhead: FORECAST_DAYS
};
var gForecastTimer;
var gForecastXSLT;
```

La variable `gForecastTimer` sera utilisée pour mettre à jour périodiquement (2 heures, soit `FORECAST_REFRESH` millisecondes) les prévisions.

Nous allons également nous doter d'un mécanisme de gestion pour le *timer* nécessaire à cette action périodique, et de fonctions pour afficher ou masquer l'indicateur, tout en nettoyant son texte interne ; en effet, comme pour l'exemple Amazon.fr, nous serons parfois amenés à lui ajouter un libellé temporaire.

Voici donc quelques fonctions utilitaires supplémentaires.

Listing 9-17 Quelques fonctions utilitaires

```
function cancelTimers() {
    if (gForecastTimer)
        window.clearInterval(gForecastTimer);
} // cancelTimers

function hideIndicator() {
    $('#indicator').hide().update('');
} // hideIndicator
```

```
function showIndicator() {
    $('#indicator').update('').show();
} // showIndicator

Ajax.Responders.register({
    onException: function(requester, e) {
        $('#indicator').hide();
        alert(e);
    }
});

Event.observe(window, 'unload', cancelTimers);
```

La fonction `cancelTimers` n'est pas destinée à être utilisée autrement qu'en déchargement de page (fermeture du navigateur, navigation ailleurs, etc.) : nous n'y prenons donc pas spécialement la peine de remettre la variable à 0 ou `null`. Notez d'ailleurs son enregistrement sur cet événement.

Remarquez aussi le même gestionnaire d'exception Ajax que précédemment, qui cachera notre indicateur et affichera l'exception dans une boîte de message.

À présent, voyons la fonction d'extraction des prévisions.

Listing 9-18 Notre fonction d'extraction des prévisions

```
function getForecast(e) {
    e && e.stop(); ①
    var url = FORECAST_TEMPLATE.evaluate(gForecastParams);
    showIndicator();
    $('#results').update(''); ②
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('#indicator').update('Mise en forme');
            var tmr = window.setTimeout(function() { ③
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, gForecastXSLT);
                $('#results').update(html);
                hideIndicator();
            }, 10);
        }
    });
} // getForecast
```

L'examen initial ① de présence d'un argument `e` est dû au fait que nous appellerons parfois la méthode directement, alors qu'elle sera aussi invoquée lors de l'envoi de notre futur formulaire de sélection de ville. Elle n'aura donc pas toujours d'objet événement associé. Si `e` n'est pas fourni, il vaut `undefined` qui est équivalent à `false`, on ne tentera donc pas d'appeler sa méthode `stop()` car l'opérateur `&&` court-circuitera l'évaluation.

Comme dans l'exemple Amazon.fr, nous commençons par vider le conteneur de résultats ②, qui contient peut-être la représentation de prévisions précédentes. Ainsi, l'utilisateur comprend immédiatement qu'une mise à jour a lieu, d'autant plus qu'il a peut-être remarqué l'indicateur en bas à gauche de la page.

Enfin, nous utilisons la même astuce qu'auparavant ③ pour donner de meilleures chances à notre libellé « Mise en forme... » d'être effectivement affiché pendant la transformation XSLT.

Il ne nous reste plus qu'à initialiser la page, ce qui se fait en deux temps :

- 1 D'abord, nous récupérons la feuille XSLT capable de transformer un résultat XML de prévisions TWC en un fragment XHTML adapté à nos besoins.
- 2 Ensuite, nous lançons la demande de prévisions.

Voici le code nécessaire.

Listing 9-19 Initialisation de la page

```
function initPage() {
    new Ajax.Request('/xsl/forecast.xsl', {
        method: 'get',
        onSuccess: function(requester) {
            gForecastXSLT = xmlParse(requester.responseText);
            getForecast(); ①
            gForecastTimer = window.setInterval(getForecast,
                ↪ FORECAST_REFRESH); ②
        }
    });
} // initPage

logging__ = false;

document.observe('dom:loaded', initPage);
```

Tout ceci doit commencer à sembler familier... On récupère la feuille XSLT, et une fois celle-ci obtenue, on la stocke dans `gForecastXSLT`, on demande ① les prévisions (vous voyez qu'ici, aucun objet événement n'est transmis en argument), et on déclenche le *timer* d'actualisation, toutes les 2 heures ②.

Remarquez la fameuse variable globale `logging_` de Google AJAXSLT, que nous mettons à `false` pour éviter d'avoir un pavé de suivi des opérations XPath/XSLT en haut à droite de la page, comme dans l'exemple précédent.

Pour chipoter, on peut argumenter que puisque la requête REST est asynchrone, `initPage` récupérera la main sans doute avant qu'on ait reçu et traité les prévisions, ce qui veut dire qu'on déclenchera le *timer* une ou deux seconde(s) trop tôt. C'est exact. Cependant sur 2 heures, dans un contexte sans contraintes précises de temps, ce n'est pas vital.

Et la feuille XSLT ?

Peut-être vous êtes-vous jeté(e) sur votre page pour tester le résultat, mais n'avez abouti qu'à du vide, voire une erreur. Eh oui ! Vous avez négligé un composant critique, que nous n'avons pas encore réalisé : la feuille XSLT. La voici.

Listing 9-20 Notre feuille XSLT pour les prévisions

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
  ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/weather"/>
    <xsl:apply-templates select="/error"/> ①
  </xsl:template>
  <xsl:template match="weather">
    <div id="weatherData"> ②
      <xsl:apply-templates select="loc"/>
      <xsl:apply-templates select="dayf"/>
    </div>
  </xsl:template>
  <xsl:template match="loc">
    <p class="cityInfo">
      <xsl:value-of select="dnam"/>
      <xsl:text> @ </xsl:text>
      <span id="cityTime"><xsl:value-of select="tm"/></span> ③
    </p>
  </xsl:template>
  <xsl:template match="dayf">
    <div class="forecast">
      <xsl:for-each select="day">
```

```

<xsl:sort select="@d"/> ④
<div class="dayForecast">
  <span class="dayName"><xsl:value-of select="@t"/></span> ③
  <xsl:element name="img">
    <xsl:attribute name="class">picto</xsl:attribute>
    <xsl:attribute name="src">icons/<xsl:value-of
      ↵ select="part[@p='d']/icon"/>.png</xsl:attribute>
    <xsl:if test="part[@p='d']/t != 'N/A'"> ⑤
      <xsl:attribute name="alt"><xsl:value-of
        ↵ select="part[@p='d']/t"/></xsl:attribute>
      <xsl:attribute name="title"><xsl:value-of
        ↵ select="part[@p='d']/t"/></xsl:attribute>
    </xsl:if>
  </xsl:element>
  <span class="temps">
    <xsl:if test="low != 'N/A'"> ⑥
      <xsl:value-of select="low"/>
    </xsl:if>
    <xsl:if test="hi != 'N/A'">
      <xsl:if test="low != 'N/A'"/></xsl:if>
      <xsl:value-of select="hi"/>
    </xsl:if>
    <xsl:if test="hi != 'N/A' or low != 'N/A'">&deg;C</xsl:if>
  </span>
  </div>
</xsl:for-each>
</div>
<p class="forecastUpdateTime">
  Derni&egrave;re mise &agrave; jour des mesures&nbsps;;
  <span id="latestUpdate"><xsl:value-of select="lsup"/></span> ③
</p>
</xsl:template>
<xsl:template match="error"> ①
  <p class="error">
    Erreur #<xsl:value-of select="err/@type"/>&nbsps;;
    <xsl:value-of select="err"/>
  </p>
</xsl:template>
</xsl:stylesheet>

```

Finalement, elle est plus légère que celle de notre exemple Amazon.fr. Je vous recommande tout de même de la récupérer dans l'archive des codes source du livre... Voici quelques remarques importantes :

- ① Lorsqu'on a commis une erreur dans la requête, ou qu'un problème réseau ou serveur a été rencontré, on reçoit parfois une réponse contenant plutôt un élément racine `error`, dont l'élément fils `err` détaille un minimum le souci. Autant utiliser la feuille XSLT pour l'afficher en cas de pépin ! Nous n'aurons de toutes façons jamais un élément racine `weatheret` une autre racine `error`, par définition.

- ② En encadrant notre sortie dans un `div` dédié, nous isolons la portion à styler du conteneur existant. Nous pouvons fournir à côté un fragment CSS dont toutes les règles sont préfixées par `#weatherData`, et être tranquilles. Ceux souhaitant plusieurs affichages utiliseraient une classe, mais je rappelle que c'est contraire aux règles d'utilisation du service de TWC...
- ③ Notez les `span` avec un ID ou une classe... Outre la possibilité de styler individuellement les données au sein d'un texte environnant, ils nous seront surtout utiles pour le reformatage partiel des informations, à l'aide d'expressions rationnelles, de `gsub` et d'objets JavaScript `Date`, comme nous allons le voir plus tard.
- ④ Voici le fameux tri explicite des journées de prévision, que je mentionnais plus haut.
- ⑤ Comme indiqué plus tôt, on ne définit les attributs `alt` et `title` de l'image que si le libellé n'est pas N/A.
- ⑥ Le traitement des N/A est ici un peu plus complexe, puisque nous avons deux données à gérer. Ce fragment n'affiche rien si nous avons deux N/A, affiche une seule température si besoin, ou les deux séparées par une barre oblique (*slash /*). Dès que nous avons une température, nous suffixons par `'C`.

Évidemment, il faut compléter la CSS.

Listing 9-21 Compléments de client.css

```
/* Affichage des prévisions */  
  
#weatherData {  
    margin: 1em auto;  
    border: 1px solid #666;  
    width: 570px;  
}  
  
p.cityInfo {  
    text-align: center;  
    font-weight: bold;  
    font-size: smaller;  
    color: navy;  
    border-bottom: 1px solid silver;  
    margin: 0 0 0.3em 0;  
    padding: 0.1em 0;  
}  
  
#cityTime {  
    color: #44f;  
}
```

```
div.forecast {  
    width: 552px;  
    height: 180px;  
    margin: 0 auto;  
}  
  
div.dayForecast {  
    float: left;  
    position: relative;  
    width: 138px;  
    height: 100%;  
}  
  
span.dayName {  
    position: absolute;  
    left: 0; top: 0; width: 128px;  
    text-align: center;  
    color: gray;  
}  
  
img.picto {  
    position: absolute;  
    left: 0; top: 32px;  
}  
  
span.temps {  
    position: absolute;  
    left: 0; bottom: 0; width: 128px;  
    text-align: center;  
    color: #444;  
}  
  
p.forecastUpdateTime {  
    text-align: center;  
    font-size: smaller;  
    color: gray;  
    border-top: 1px solid silver;  
    margin: 0.3em 0 0 0;  
    padding: 0.1em 0;  
}  
  
p.error {  
    width: 60ex;  
    margin: 1em auto;  
    padding: 1em;  
    border: 1px solid maroon;  
    background: #fdd;  
    color: red;  
    text-align: center;  
}
```

Cette fois ça y est ! Vous pouvez rafraîchir (pensez à contourner le cache), cela devrait marcher (figure 9-8).

Figure 9-8

Nos prévisions à 4 jours pour Paris



Les petites touches finales

Il nous reste à apporter quelques petits ajustements qui font, pourtant, toute la différence... Si vous observez bien la figure 9-8 (ou votre écran), vous remarquerez que :

- L'heure locale est en anglais (*damned!*!).
- Les noms des jours aussi (re-damned!).
- Et l'heure de dernière mise à jour des mesures, n'en parlons pas !
- Qui plus est, l'heure locale « n'avance » pas comme une horloge, alors que nous souhaitions en réaliser une.

Il est temps de nous attaquer à tout cela. Commençons par modifier les textes vers des valeurs françaises (ou, dans le cas des dernières mesures, et par paresse, des valeurs *dans la langue active du navigateur*).

Pour ce faire, nous allons utiliser le même genre d'algorithme que pour notre exemple Amazon.fr, avec une fonction `adjustData`, des expressions rationnelles, des appels à la méthode `gsub` ajoutée aux `Strings` par Prototype... mais aussi quelques astuces spécifiques à cet exemple.

Voici déjà quelques « constantes » supplémentaires à ajouter en début de script.

Listing 9-22 Nouvelles « constantes » en début de script

```

DAY_NAMES = {
    'Monday': 'lundi', 'Tuesday': 'mardi',
    'Wednesday': 'mercredi', 'Thursday': 'jeudi',
    'Friday': 'vendredi', 'Saturday': 'samedi',
    'Sunday': 'dimanche'
};

RE_LOCALTIME = '(<span id="cityTime">)(.*?)(</span>)';
RE_LATESTUPDATE = '(<span id="latestUpdate">)(.*?) Local Time(</span>)';
RE_DAYNAME = '(<span class="dayName">)(.*?)(</span>)';

```

L'objet DAY_NAMES est une première astuce : en nommant ses propriétés d'après les noms anglais des jours, nous pourrons faire correspondre un nom français à chaque nom anglais, rien qu'en faisant DAY_NAMES[variableDeNomAnglais] (souvenez-vous que l'opérateur [] permet d'accéder aux propriétés).

Ici, nous aurions pu nous passer des apostrophes autour des noms des propriétés, mais cela n'est pas toujours vrai (certaines langues utilisent dans leurs noms de jours des caractères invalides dans un identifiant JavaScript) : autant prendre ses précautions.

Côté expressions rationnelles, le quantificateur *? est la version **réticente** de * : il signifie « prend la plus petite quantité permettant toujours de satisfaire l'ensemble de l'expression ». Dans le cas qui nous occupe, on est ainsi sûr de s'arrêter au premier `` rencontré, plutôt qu'au dernier (ce qui poserait de gros problèmes). Par conséquent, nos groupes .*? contiennent tout ce qui se trouve entre le `>` fermant et le premier `` rencontré : en somme, le contenu du `span`.

Nous aurons aussi besoin d'un nouveau *timer* pour gérer l'horloge, et d'une autre variable, un objet Date, pour l'heure de cette horloge. Au passage, nous ajusterons donc `cancelTimers` pour prendre en compte ce second *timer*.

```

var gLocalTime;
var gLocalTimeTimer;

function cancelTimers() {
    if (gForecastTimer)
        window.clearInterval(gForecastTimer);
    if (gLocalTimeTimer)
        window.clearInterval(gLocalTimeTimer);
} // cancelTimers

```

Il est temps d'écrire notre fameuse fonction `adjustData`. Son code est intéressant à comprendre.

Listing 9-23 Notre fonction d'ajustement de XHTML, adjustData

```
function adjustData(html) {
    // L'heure locale courante du lieu affiché
    html = html.gsub(RE_LOCALTIME, function(match) {
        gLocalTime = new Date('1/1/1970 ' + match[2]); ①
        gLocalTime.setSeconds(new Date().getSeconds());
        return match[1] + gLocalTime.toLocaleTimeString() + match[3]; ②
    });
    // L'heure de dernière mise à jour
    html = html.gsub(RE_LATESTUPDATE, function(match) {
        var lsup = new Date(match[2]);
        if (lsup.getFullYear() < 1980) ③
            lsup.setFullYear(lsup.getFullYear() + 100);
        return match[1] + lsup.toLocaleString() + match[3]; ②
    });
    // Jours de la semaine
    html = html.gsub(RE_DAYNAME, function(match) {
        return match[1] + DAY_NAMES[match[2]] + match[3];
    });
    return html;
} // adjustData
```

Voyons ensemble ses quelques points saillants :

- ➊ Il n'est pas possible de construire un objet `Date` juste sur la base d'un texte horaire : il faut une partie date. Dans la mesure où nous ne nous intéressons cependant pas à la date (nous ne l'afficherons pas), nous en prenons une quelconque. Ici, nous avons opté pour le célèbre 1^{er} janvier 1970 : dans la plupart des systèmes, la représentation zéro pour une date correspond au 1^{er} janvier 1970 à 00:00:00 GMT. Remarquez l'espace en fin de texte, sans quoi la chaîne obtenue n'est plus valide (`01/01/19704:18 PM...`).

Autre point important : comme nous allons afficher les secondes (après tout, c'est une horloge détaillée...), il nous faut autre chose que les zéro secondes que va produire `Date.parse`. Quel que soit le fuseau horaire, les secondes sont les mêmes (d'accord, il existe *un* cas de figure où cela ne sera pas vrai) : nous utilisons donc les secondes locales.

Enfin, remarquez qu'on stocke l'heure de l'horloge dans la variable *globale* `gLocalTime`, que nous pourrons utiliser ensuite pour mettre à jour l'horloge.

- ➋ Les appels à `toLocaleTimeString()` et `toLocaleString()` demandent au navigateur de fournir une représentation textuelle complète adaptée aux conventions de la langue active. La première n'affiche que l'heure, l'autre affiche aussi la date. Suivant le navigateur, on aura la langue de l'environnement ou non, le code de fuseau horaire ou non...

③ Les implémentations de JavaScript n'ont pour la plupart pas de « pivot » sur l'interprétation des dates. Du coup, une date à partir de 2000 exprimée sur deux chiffres va utiliser l'ancienne interprétation, et régresser de cent ans ! Nous corrigeons manuellement ce cas de figure, qui nous concerne d'ailleurs, en supposant un pivot à 1980.

Que nous reste-t-il à faire ? Tout d'abord, il va nous falloir écrire une fonction de mise à jour de l'horloge, qui sera appelée toutes les secondes.

Listing 9-24 Gestion de l'horloge

```
function updateLocalTime() {
    gLocalTime.setTime(gLocalTime.getTime() + 1000);
    $('#cityTime').update(gLocalTime.toLocaleTimeString());
} // updateLocalTime
```

Ensuite, il nous faut appeler `adjustData` : pour l'instant, le code ne s'en sert pas. Et en prévision des rafraîchissements de prévisions (automatiques ou explicites, avec le formulaire que nous allons réaliser dans un instant), nous devons désactiver le *timer* d'horloge avant rafraîchissement, et le réactiver une fois le nouveau fragment XHTML affiché (et encore, à condition qu'il ne s'agisse pas d'une erreur, et qu'on dispose donc d'un élément d'ID `cityTime`).

Tout ceci a lieu dans la nouvelle version de `getForecast`.

Listing 9-25 Notre nouvelle fonction `getForecast`

```
function getForecast(e) {
    e && e.stop();
    if (gLocalTimeTimer)
        window.clearInterval(gLocalTimeTimer);
    var url = FORECAST_TEMPLATE.evaluate(gForecastParams);
    showIndicator();
    $('#results').update('');
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('#indicator').update('Mise en forme');
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsiltProcess(data, gForecastXSLT);
                $('#results').update(adjustData(html));
                if ($('#cityTime'))
                    gLocalTimeTimer = window.setInterval(updateLocalTime, 1000);
            }, 1000);
        }
    });
}
```

```
        hideIndicator();
    }, 10);
}
} );
} // getForecast
```

Un petit rafraîchissement strict, et voici l'écran de la figure 9-9.

Figure 9-9

Un affichage localisé, et une horloge qui fonctionne



C'est déjà beau, je vous sens ému(e), mais ce petit code n'est rien à côté de ce qui nous attend à la prochaine section.

Rechercher un lieu

Nous allons ajouter une dernière fonctionnalité à cette page si sympathique : la possibilité de changer de lieu. Et pour cela, nous allons devoir requérir une autre ressource de l'API REST de TWC.

Cependant, cette requête exige que le nom soit assez complètement tapé pour avoir des correspondances exactes (par exemple, « Pa » ne renverra rien : pour avoir des suggestions de « Paris », il faut taper « Paris »... on a alors quatorze résultats !). On utilisera une complétion automatique de texte Ajax, pour que ce soit plus joli et plus intuitif. Cependant cette optique va nous amener à réaliser un code qui, s'il n'est pas très compliqué, va néanmoins vous remplir de fierté.

Préparons le terrain

Commençons par définir le formulaire, en début de corps de page dans `index.html`, et ajouter les styles correspondants dans `client.css`.

Listing 9-26 Le formulaire inséré dans `index.html`

```
<h1>Fébô&nbsp;? Fépabô&nbsp;?</h1>

<form id="cityForm">
  <p>
    <label for="edtCity" accesskey="V">Ville (nom anglais)</label>
    <input type="text" id="edtCity" name="city" value="Paris, France" />
    <input type="submit" id="btnDisplay" value="Afficher"
           disabled="disabled" />
    <div id="city_suggestions" class="autocomplete"></div>
  </p>
</form>

<div id="results"></div>
```

Remarquez que nous avons désactivé le bouton d'envoi, comme pour Amazon.fr : nous ne le réactiverons qu'une fois chargée la feuille XSLT (nécessaire à la transformation des résultats de recherche).

Listing 9-27 Les styles pour le formulaire, dans `client.css`

```
#cityForm {
  width: 34em;
  border: 1px solid gray;
  margin: 1em auto;
  padding: 0.5em;
}

#cityForm p {
  position: relative;
}

#edtCity {
  position: absolute;
  left: 10em;
  width: 17em;
}

#btnDisplay {
  position: absolute;
```

```
    left: 28em;
    width: 6em;
}

div.autocomplete {
    position: absolute;      /* Histoire d'être explicite... */
    width: 250px;           /* Sera ajusté par script.aculo.us */
    background-color: white;
    border: 1px solid #888;
    margin: 0px;
    padding: 0px;
    z-index: 42;
}

div.autocomplete ul {
    list-style-type: none;
    margin: 0px;
    padding: 0px;
}

div.autocomplete ul li {
    list-style-type: none;
    display: block;
    margin: 0;
    cursor: default;
    /* Et quelques finasseries... */
    padding: 0.1em 0.5ex;
    font-family: sans-serif;
    font-size: 90%;
    color: #444;
    height: 1.5em;
    line-height: 1.5em;
    overflow: hidden;
}

div.autocomplete ul li span.informal { ❶
    display: none;
}
/* Rappel : script.aculo.us active une classe 'selected'
   lorsqu'un élément est sélectionné */
div.autocomplete ul li.selected {
    background-color: #ffb;
```

Remarquez que nous utilisons exactement le même bloc de styles basiques que dans notre exemple de complétion simple au chapitre 8. Ces styles viennent d'ailleurs directement des suggestions du wiki documentaire de script.aculo.us.

Nous attirons toutefois votre attention sur la déclaration ❶. Nous avons vu que script.aculo.us ignorera tout texte contenu dans un élément de classe `informal` lorsqu'il extraira de l'élément `li` le texte à placer dans le champ de saisie. En général, nous affichons néanmoins de tels textes dans la liste des suggestions. Ici non, nous verrons pourquoi tout à l'heure.

Nous allons aussi ajouter à notre script `client.js` une « constante » pour composer l'URL de requête, ainsi qu'une variable pour recevoir la feuille XSLT dont nous aurons besoin.

Listing 9-28 Ajouts à notre script client.js

```
LOCATION_TEMPLATE = new Template(  
    'http://xoap.weather.com/search/search?where=#{name}');  
  
var gSearchXSLT;
```

Regardons à présent à quoi ressemblent notre requête et le XML de réponse. La requête est simple : l'URL `http://xoap.weather.com/search/search` accepte un paramètre `where` avec le texte saisi. Par exemple :

```
http://xoap.weather.com/search/search?where=paris
```

Le XML résultat est également trivial. Voici le résultat pour « Moscow ».

Listing 9-29 Résultat XML d'une recherche de lieu sur « Moscow »

```
<search ver="2.0">  
  <locid="RSXX0063" type="1">Moscow, Russia</loc>  
  <loc id="USAR0390" type="1">Moscow, AR</loc>  
  <loc id="USID0170" type="1">Moscow, ID</loc>  
  <loc id="USIA0594" type="1">Moscow, IA</loc>  
  <loc id="USKS0396" type="1">Moscow, KS</loc>  
  <loc id="USMI0572" type="1">Moscow, MI</loc>  
  <loc id="USOH0629" type="1">Moscow, OH</loc>  
  <loc id="USPA1102" type="1">Moscow, PA</loc>  
  <loc id="USTN0346" type="1">Moscow, TN</loc>  
  <loc id="USTX0919" type="1">Moscow, TX</loc>  
  <loc id="USVT0151" type="1">Moscow, VT</loc>  
</search>
```

Comme vous pouvez le constater, les américains ont décidément eu à cœur de semer des noms de villes européennes dans un état sur quatre... Il en va de même pour Florence, London... Et Paris, évidemment.

En tout cas, le XML est trivial. Le seul souci, c'est qu'il est inutilisable avec notre complétion automatique Ajax : celle-ci exige une structure `ul/li`, alors que nous avons ici du `search/loc`. Et `Ajax.AutoComplete` exploite directement le contenu récupéré. Comment faire ?

Éblouissez vos amis avec Ajax.XSLTCompleter

Nous commencerons par utiliser certaines options avancées de `Ajax.AutoComplete`, que nous n'avons guère eu l'occasion d'employer jusque-là :

- `callback`, qui permet de fournir une méthode chargée de transformer les paramètres de requête calculés par défaut ; nous l'utiliserons pour remplacer le simple nom de lieu par notre fameux paramètre `url` avec l'URL complète de requête au service REST, dont a besoin `/xmlProxy`.
- `afterUpdateElement`, qui nous donne la main une fois une sélection établie par l'utilisateur. Ici, avoir le nom de la ville intéresse l'utilisateur, mais nous, nous avons besoin du code interne (par exemple, FRXX0076). Il faudra le stocker dans le champ `code` de notre variable `gForecastParams`, pour que `getForecast` l'utilise dans sa requête.

Ces deux options ne résolvent toutefois pas le problème de transformation à la volée d'un format de document (le XML résultat de TWC) en un fragment XHTML `ul/li`.

Pour cela, nous allons devoir véritablement mettre les mains dans le cambouis, en créant **une sous-classe de `Ajax.AutoComplete`** ! Oui, vous avez bien lu : nous allons étendre une classe Prototype, et d'une manière non triviale.

L'idée est simple : nous allons créer un objet « héritant » de `Ajax.AutoComplete`, qui définira sa propre fonction de rappel `onComplete`, à la place de l'existante (qui existe en interne dans le prototype parent). Appelée en fin de récupération de réponse, cette méthode pourra analyser le texte de la réponse pour produire une grappe XML, et appliquer une transformation. Il ne lui reste alors plus qu'à appeler la méthode-clef chez les objets de complétion automatique : `updateChoices`, en lui passant le fragment XHTML. Et le tour est joué ! Dans `Ajax.AutoComplete`, on appelle `updateChoices` en lui passant directement le `responseText`. Ici, nous interceptons le résultat pour le transformer.

Nous allons évidemment rendre cet objet réutilisable, en ajoutant une option `xslt` à la liste des options existantes. Cerise sur le gâteau, cette option pourra être soit du texte, soit un DOM. Dans le premier cas, on l'analysera à l'instanciation pour produire un DOM.

Le code résultant fait appel à quelques morceaux ardu斯 de JavaScript, en particulier si vous n'avez pas bien suivi l'héritage par prototypes au chapitre 2. Toutefois, lorsque

vous aurez compris ce code, vous serez bien avancé(e) sur le chemin qui mène au statut de gourou JavaScript !

Voici notre nouvel objet, à ajouter dans `client.js`.

Listing 9-30 Notre objet Ajax.XSLTCompleter

```
Ajax.XSLTCompleter = Class.create(Ajax.AutoComplete, {
  initialize: function($super, element, update, url, options) { ①
    $super(element, update, url, options); ②
    this.options.onComplete = this.onComplete.bind(this);
    var options = args[2] || {}
    if ('string' == typeof this.options.xslt)
      this.options.xslt = xmlParse(this.options.xslt);
  },
  onComplete: function(request) {
    var data = xmlParse(request.responseText);
    var html = xsltProcess(data, this.options.xslt);
    this.updateChoices(html); ③
  }
});
```

Et voilà le travail ! Quelques précisions tout de même :

- ① Dans les classes Prototype (les classes obtenues en appelant `Class.create()`), le « constructeur » s'appelle `initialize`. Pour rappel, Prototype utilise la convention de premier argument `$super` pour indiquer qu'on souhaite y récupérer la version héritée de la méthode afin de l'appeler ensuite.
- ② Le premier argument, `$super`, représente la version héritée de `initialize`, à laquelle on passe les arguments d'origine.
- ③ C'est cet appel qui va mettre à jour la liste des suggestions et l'afficher automatiquement.

Cet objet étant prêt, il nous reste à disposer d'une feuille de styles, et à utiliser tant l'objet que la feuille. Voici déjà le document, `search.xsl`, à mettre dans `docroot/xsl`.

Listing 9-31 Notre feuille xsl/search.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="/search"/>
  </xsl:template>
```

```

<xsl:template match="search">
  <ul>
    <xsl:apply-templates select="loc"/>
  </ul>
</xsl:template>
<xsl:template match="loc">
  <li>
    <span class="informal"><xsl:value-of select="@id"/></span> ①
    <xsl:value-of select=". . ."/>
  </li>
</xsl:template>
</xsl:stylesheet>

```

Brancher les composants ensemble

C'est maintenant la dernière ligne droite de notre exemple : nous allons insérer du code supplémentaire au début de notre fonction `initPage`. Ce code a trois fonctions :

- 1 Associer `getForecast` à l'envoi du formulaire, comme cela a été prévu dès le départ.
- 2 Requêter la feuille XSLT
- 3 Une fois celle-ci obtenue, initialiser la complétion automatique sur le champ du formulaire, avec quelques paramètres spéciaux...

Voici donc le nouveau code, figurant au tout début de `initPage` (j'ai laissé les lignes alentour pour vous aider à vous repérer).

Listing 9-32 Notre initialisation finalisée

```

function initPage() {
  $('#cityForm').observe('submit', getForecast); ①
  new Ajax.Request('/xsl/search.xsl', { ②
    method: 'get',
    onSuccess: function(request) {
      gSearchXSLT = xmlParse(request.responseText);
      new Ajax.XSLTCompleter('edtCity', 'city_suggestions',
        '/xmlProxy', { ③
          method: 'get',
          paramName: 'foo', ④
          indicator: 'indicator', ⑤
          minChars: 2, ⑥
          xslt: gSearchXSLT, ⑦
          callback: function(e, entry) { ⑧
            var url = LOCATION_TEMPLATE.evaluate(
              { name: entry.substring(4) });
            return 'url=' + encodeURIComponent(url);
          },
        }
      );
    }
  });
}

```

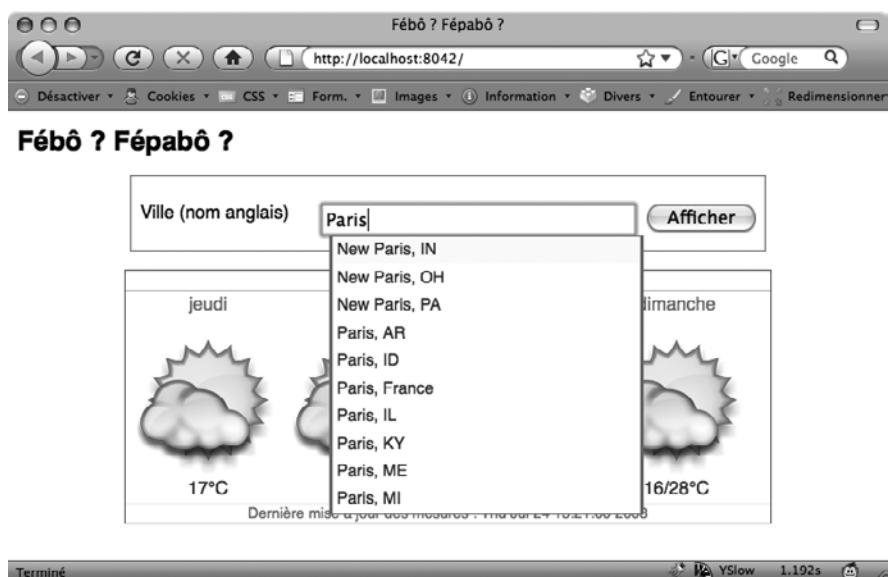
```
        afterUpdateElement: function(e, elt) {
            gForecastParams.code = elt.firstChild.firstChild.nodeValue; ⑨
        }
    });
    $('#btnDisplay').disabled = false; ⑩
}
});
new Ajax.Request('/xsl/forecast.xsl', {
```

Vu la richesse de ce code, je vous gratifie d'un bon nombre de remarques :

- ① Nous associons, évidemment, `getForecast` à l'envoi du formulaire. Souvenez-vous du `Event.stop` au début de `getForecast` : cela empêchera l'envoi normal, au profit d'une recherche Ajax. Celle-ci utilisera `gForecastParams.code`, qui aura été mis à jour par la fonction de rappel `afterUpdateElement` de la complétion automatique.
- ② Voilà notre chargement de feuille XSLT pour transformer les résultats de recherche en fragment compatible (`ul/li`) avec `Autocompleter.Base`, le mécanisme générique de complétion automatique.
- ③ Évidemment, nous utilisons à nouveau notre `/xmlProxy` à tout faire...
- ④ Nous pourrions nous en passer, mais comme le seul paramètre que nous souhaitons est en fait le paramètre `url` que va synthétiser notre fonction de rappel `callback`, autant bien indiquer que le paramètre d'origine ne sert à rien côté serveur (il ne sera même pas envoyé, car `callback` ne le préserve pas).
- ⑤ Facilité toujours : vous souveniez-vous que l'option `indicator` affiche et masque toute seule notre indicateur ? Puisqu'ici nous ne nous soucions pas de jouer sur son libellé, nous pouvons utiliser cette option.
- ⑥ TWC envoie une erreur si nous effectuons une recherche avec un nom d'un seul caractère pour le paramètre `where`. Il est vrai que ce n'est pas très utile. Disons 2 au minimum.
- ⑦ Hourra ! Voici notre option personnalisée ! Nous lui transmettons directement notre variable globale qui contient la feuille XSLT à utiliser. Comme le code courant s'exécute depuis le `onSuccess` du chargement de cette XSLT, on sait que la variable est prête.
- ⑧ La fonction de rappel `callback` reçoit le champ de saisie et le paramètre correspondant à la saisie en cours, prévu pour envoi côté serveur. Ici, c'est donc par exemple '`foo=blah`' (pour une saisie de `blah`, et parce que nous avons dit `paramName: 'foo'`).

Nous éliminons les 4 premiers caractères pour récupérer le texte encodé, que nous injectons dans notre modèle d'URL. Nous encodons le tout et nous préfixons par '`url='`', le paramètre attendu par `/xmlProxy`.

Figure 9-10
Les lieux possibles
pour « Paris »



- ⑨ Relisez la feuille XSLT : chaque `li` contient d'abord un premier élément fils `span`, qui contient un seul élément fils de type texte. Pour accéder à la valeur de ce texte, on a donc `elt` (l'élément `li`), un premier `firstChild` (le `span`), un second (le nœud texte dans le `span`), et un `nodeValue` (le texte lui-même).
- ⑩ N'oublions pas, après tout ça, de finir de réagir au chargement de la feuille XSLT en activant le bouton du formulaire !

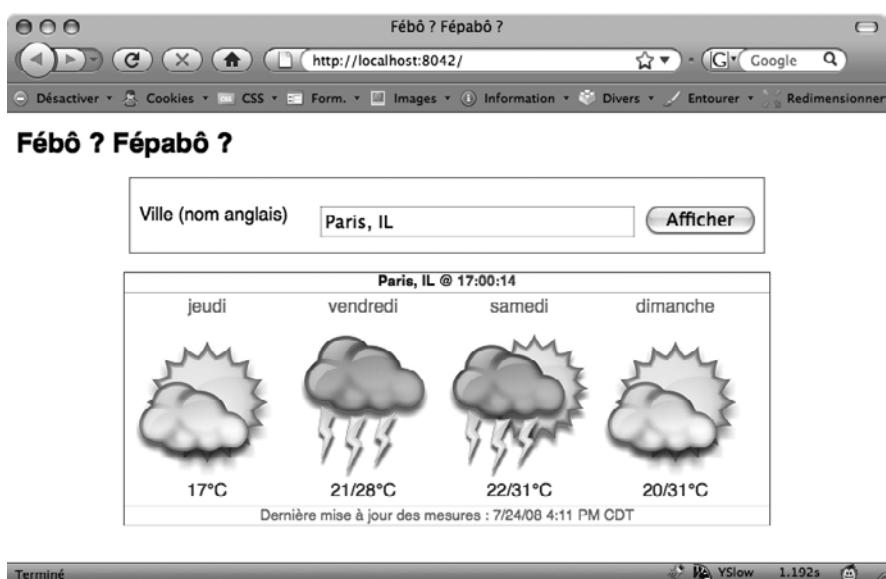
Cette fois vous y êtes : vous pouvez faire un rafraîchissement strict, et taper autre chose dans la zone, par exemple « Paris ». Vous devriez voir rapidement apparaître une liste de possibilités (figure 9-10).

Remarquez « Paris/Charles de Gaulle, France » : les services météo traitent généralement à part les aéroports. Choisissons par exemple « Paris, IL » (Illinois, USA), et validons le formulaire (sous Opera et Safari, il est validé parfois directement), ce qui donne l'écran de la figure 9-11.

Notez l'heure locale. L'Illinois, c'est 7 heures avant la France...

Allez, il est l'heure d'appeler les collègues et de montrer fièrement vos prouesses...

Figure 9–11
Le temps dans
l'Illinois



Gérer des images chez Flickr

Flickr, récemment acquis par Yahoo!, est le site de référence pour publier, partager, cataloguer et diffuser des photos en ligne. Fort d'une direction technique de haut vol, le site propose une API très complète, accessible en REST, en XML-RPC ou en SOAP (rien que dans l'ordre de ces formats, qui est celui figurant partout sur le site, on sent que la couche technique va dans le bon sens...).

Cette API est merveilleusement documentée, de façon très claire et détaillée. Le point d'entrée de la documentation est <http://www.flickr.com/services/api/> (tout simplement). Chaque méthode est décrite avec son synopsis, le détail de ses arguments, ses contraintes en authentification, un exemple du contenu de réponse, et la liste détaillée des codes d'erreur avec leurs causes !

C'est déjà fabuleux, mais on trouve en plus à chaque fois un lien sur l'explorateur API, qui permet de tester immédiatement, interactivement, un appel à la méthode (pour peu qu'on soit connecté et qu'on dispose d'une clef API). Un vrai bonheur...

Par ailleurs, de nombreuses bibliothèques pour cette API sont disponibles, pour la plupart des langages et plates-formes (citons notamment, par ordre alphabétique : ActionScript, Delphi, Java, .NET, PHP, Python et Ruby).

Dans ce dernier exemple, nous allons afficher un jeu de photos publié par un utilisateur de Flickr (et pas des moindres : Tristan Nitot, le président de Mozilla Europe

en personne), afficher les vignettes, et en cliquant dessus, afficher la photo plus grande, en tant que lien vers la taille originale, ainsi que les commentaires de la photo.

Nous n'utiliserons en revanche pas de méthodes modifiantes (ajout de commentaire, etc.) car elles nécessitent un mécanisme d'authentification mal adapté à nos conditions de test. Ce n'est pas tant qu'il faille calculer des *hashes* MD5 (des bibliothèques JavaScript font cela très vite), mais qu'il faille fournir à Flickr une URL *accessible sur le Web*, connectée à notre application... Cela ferait plusieurs allers-retours consécutifs, et deviendrait vite trop épais pour ce chapitre.

Obtenir une clef API chez Flickr

Il faut d'abord avoir un compte chez Flickr, ce qui signifie aujourd'hui un compte chez Yahoo!. Si vous n'avez pas de compte, créez-en un sur <http://www.flickr.com/signup> :

- 1 Choisissez le lien Sign Up sur la droite.
- 2 Saisissez les champs nécessaires. Attention, le code de confirmation est sensible aux majuscules/minuscules. Utilisez une adresse électronique valide, car vous y recevrez un lien d'activation de compte.
- 3 Récupérez votre courriel de bienvenue et suivez son lien d'activation.
- 4 Choisissez un nom de compte Flickr, qui peut être différent de votre nom de compte Yahoo!
- 5 Activez le bouton Sign In.

À présent que vous êtes connecté(e), demandez une clef en allant sur <http://www.flickr.com/services/api/key.gne> :

- 1 Vérifiez votre nom et votre adresse électronique.
- 2 Si vous demandez ici une clef pour une utilisation commerciale, cochez le bouton radio correspondant. Attention toutefois, le service dans un contexte commercial est généralement payant...
- 3 Décrivez rapidement l'utilisation que vous allez faire de la clef (par exemple, « I'm going to use it to explore your services while reading a book that uses Flickr REST examples »).
- 4 Cochez la case I agree to Flickr APIs Terms of Use.
- 5 Activez le bouton Apply.
- 6 Et voilà ! Votre clef s'affiche. Copiez-collez-la en lieu sûr.

Prenez le temps de lire le contrat d'utilisation de l'API. Il s'agit surtout de respecter les règles en vigueur sur Flickr, de respecter la propriété intellectuelle, et de réagir avec diligence aux demandes des propriétaires de photos que vous exposez à travers

votre application. La page décrivant les conditions d'utilisation est plutôt claire, ne manquez pas de la parcourir.

Format général des requêtes et réponses REST chez Flickr

Vous pouvez vérifier que votre clef est active en utilisant la méthode (l'opération, si vous préférez) `flickr.test.echo`, comme ceci, en précisant évidemment votre clef réelle :

```
http://api.flickr.com/services/rest/
?method=flickr.test.echo&opinion=PasMalCeBouquin&api_key=VOTRE_CLEF_ICI
```

Vous devez obtenir un résultat XML comme celui-ci :

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<method>flickr.test.echo</method>
<opinion>PasMalCeBouquin</opinion>
<api_key>VOTRE_CLEF_ICI</api_key>
</rsp>
```

D'une manière générale, une requête REST Flickr utilise toujours le point d'accès suivant :

```
http://api.flickr.com/services/rest/
```

L'opération et le type de ressource voulus sont encodés dans la **méthode**, un paramètre `method` dont le nom utilise une syntaxe hiérarchique à points, comme les méthodes `flickr.blogs.getList`, `flickr.groups.browse` ou `flickr.photos.addTags`.

On doit obligatoirement préciser la clef API à travers le paramètre `api_key`. Le reste dépend de la méthode. Flickr n'utilise que les verbes HTTP GET et POST, ce qui n'est pas toujours sémantiquement approprié, mais ce n'est pas très grave.

Une réponse REST Flickr a deux formats possibles. Soit l'appel a réussi, et on obtient ce type de contenu.

Listing 9-33 Une réponse REST Flickr en cas de succès

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
Contenu spécifique à la méthode ici
</rsp>
```

Soit l'appel a échoué, et on obtient ce type de contenu.

Listing 9-34 Une réponse REST Flickr en cas d'échec

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="fail">
    <err code="code-erreur" msg="message-erreur" />
</rsp>
```

Il est donc facile de détecter un problème rien qu'avec l'attribut `stat` de l'élément racine `rsp` (chemin XPath : `/rsp/@stat`).

Préparer le terrain

Commencez par créer un répertoire de travail `flickr`, et copiez-y le `serveur.rb` d'un des deux exemples précédents. Recréez aussi `docroot`, son sous-répertoire `ajaxslt` et un sous-répertoire `xsl` vide. Vous pouvez aussi préparer des squelettes pour `index.html`, `client.css`, `client.js`, et les *quatre* feuilles XSLT que nous allons utiliser : `photoset.xsl`, `photoset_owner.xsl`, `photoset_photos.xsl` et `photo.xsl`.

Notre page étant générée presque entièrement suite à des requêtes Ajax, nous avons un fichier `index.html` au corps plutôt simple.

Listing 9-35 Notre page, index.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  xml:lang="fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
        charset=iso-8859-15" />
    <title>Les jeux de photo de Tristan&#8230;</title>
    <link rel="stylesheet" type="text/css" href="client.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
        src="scriptaculous.js?load=effects,dragdrop"></script>
    <script type="text/javascript" src="ajaxslt/misc.js"></script>
    <script type="text/javascript" src="ajaxslt/dom.js"></script>
    <script type="text/javascript" src="ajaxslt/xpath.js"></script>
    <script type="text/javascript" src="ajaxslt/xslt.js"></script>
    <script type="text/javascript" src="client.js"></script>
</head>
<body>

<div id="indicator" style="display: none;"></div>
```

```
<div id="results"></div>

<div id="photo" style="position: absolute; display: none">
  <span id="photo-closer">[x]</span>
  <div id="photo-contents"></div>
</div>

</body>
</html>
```

Remarquez le chargement de `dragdrop.js`, qu'il vous faudra donc récupérer dans `script.aculo.us` (en revanche, nous n'utilisons plus `controls.js`). Le corps de la page sera dans `results`, tandis que `photo` est un conteneur destiné à afficher, en surplomb du contenu normal, une photo individuelle. Une pseudo-case de fermeture figurera en haut à droite.

Voyons à présent une première version de la feuille de styles.

Listing 9-36 Le début de notre feuille client.css

```
* {
  font-size: 1em;
}

body {
  font-family: sans-serif;
}

#indicator {
  position: absolute;
  left: 10px; top: 10px; height: 16px; width: 20em;
  font-size: 14px; line-height: 16px;
  background: url(spinner.gif) no-repeat;
  padding-left: 20px;
  color: gray;
}

#results {
  margin-top: 36px;
}
```

Rien de bien méchant. Remarquez que cette fois, on positionne l'indicateur en haut de la page, toujours afin qu'il ne sursaute pas lorsqu'on en modifie le contenu...

Notre script `client.js` reprend par ailleurs quelques fragments désormais classiques, que nous compléterons par la suite.

Listing 9-37 Squelette initial de notre client.js

```
function initPage() {
    new Draggable('photo', {});
} // initPage

Ajax.Responders.register({
    onException: function(requester, e) {
        $('indicator').hide();
        alert(e);
    }
});

logging__ = false;

document.observe('dom:loaded', initPage);
```

Remarquez qu'on peut déjà définir le conteneur photo comme étant déplaçable par glissement.

Chargement centralisé parallèle des feuilles XSLT

Nous commençons à avoir de nombreuses feuilles à charger. Le code que nous avions écrit jusqu'ici les chargeait séquentiellement, et au prix d'imbrications parfois complexes de `new Ajax.Request...`. Pour éviter cela, nous pouvons écrire un petit objet apte à charger autant de feuilles XSLT qu'on le souhaite, en parallèle si plus est.

Cet objet pourrait avoir un *hash* nommé `sheets`, dont les clefs seraient les noms de ses propriétés finales (qui contiendront le DOM des feuilles), et les valeurs seraient les « noms racines » des fichiers. On supposerait en effet que toutes les feuilles seraient dans le même répertoire, et aurait la même extension. Cet objet pourrait charger ces feuilles en parallèle, exposer le DOM obtenu pour chacune sous forme de propriété, et prévenir une fonction de rappel quand toutes les feuilles ont été chargées.

Tout ceci semble complexe, mais en réalité, cela n'utilise rien de très difficile. Regardez l'objet anonyme suivant, que nous allons placer au début de notre fichier de script `client.js`.

Listing 9-38 Notre chargeur parallèle de feuilles XSLT

```
var gXSLT = {
    _loaded: false, ①

    sheets: { ②
        photoset: 'photoset',
        owner:    'photoset_owner',
```

```

        photos:  'photoset_photos',
        photo:   'photo'
    },

    load: function(onComplete, expose) { ③
        if (this._loaded)
            return;
        this._exposeSheets = false !== expose; ④
        this._onComplete = onComplete;
        this._loadCount = 0;
        this._sheetCount = Object.keys(this.sheets).length; ⑤
        for (var s in this.sheets)
            this._loadSheet(s);
    },

    _loadSheet: function(sheet) {
        new Ajax.Request('/xsl/' + this.sheets[sheet] + '.xsl', {
            method: 'get',
            onSuccess: this._sheetLoaded.bind(this, sheet)
        });
    },

    _sheetLoaded: function(sheet, requester) {
        this.sheets[sheet] = xmlParse(requester.responseText);
        if (this._exposeSheets) {
            if (undefined !== this[sheet])
                throw('Cannot expose sheet ' + sheet +
                    ': property exists.');
            this[sheet] = this.sheets[sheet]; ⑥
        }
        if (++this._loadCount >= this._sheetCount) {
            this._loaded = true;
            (this._onComplete || Prototype.emptyFunction)();
        }
    }
}; // gXSLT

```

Voyons certains fragments de plus près...

- ① Ce drapeau nous permettra d'ignorer des appels multiples à `load`.
- ② C'est là qu'on définit les feuilles à charger. Nous avons ici, par exemple, une feuille de nom racine `photoset_photos` dont le DOM devra être exposé dans une propriété `photos`.
- ③ C'est cette fonction qui est destinée à être appelée de l'extérieur : elle déclenche le chargement parallèle. On lui transmet une fonction de rappel (optionnelle), et éventuellement un drapeau indiquant si on doit exposer les DOM obtenus sous forme de propriétés de l'objet `gXSLT`.

- ④ En utilisant la différence stricte `!==`, nous obtenons `true` dans tout autre cas que `false` spécifiquement (donc `null` ou `undefined` donnent `true`), ce qui fait que `true` est la valeur par défaut.
- ⑤ Petite astuce pour obtenir le nombre de propriétés dans `sheets`, donc de feuilles à charger : on appelle la méthode `Object.keys()` de Prototype, qui renvoie un tableau des noms de propriétés, puis on appelle `length`.
- ⑥ Voilà comment exposer le DOM sous forme de propriété directe (`gXSLT.photo` plutôt que `gXSLT.sheets.photo`, qui marche aussi, ceci dit). Remarquez qu'on hurle si une telle propriété existe déjà.
- ⑦ Voici un code typique de Prototype ! Cela nous évite un `if` pour tester qu'on nous a bien transmis une fonction de rappel. C'est la raison d'être de `Prototype.emptyFunction`.

Obtenir les informations du jeu de photos

Il est temps à présent d'attaquer le requêtage. Nous allons commencer par récupérer les informations du jeu de photos qui nous intéresse. Toutes les requêtes Flickr ayant la même structure de base, nous allons utiliser quelques « constantes » de bon aloi.

Listing 9-39 Définition du requêtage

```
FLICKR_API_KEY = 'VOTRE_CLEF_ICI';
FLICKR_ENDPOINT = 'http://api.flickr.com/services/rest/';
FLICKR_TEMPLATE = new Template(
  FLICKR_ENDPOINT + '?method=flickr.{method}&api_key=' +
  FLICKR_API_KEY + '#{extraArgs}');
PHOTOSET_ID = 1603279; // Ou 1583292...
```

Comme toutes les méthodes Flickr commencent par « `flickr.` », nous allons en dispenser le code qui suit, en utilisant statiquement ce préfixe. Remarquez que nous pourrons préciser les arguments spécifiques à la méthode à l'aide d'une propriété `extraArgs`. Nous devrons donc passer à `FLICKR_TEMPLATE` un objet avec deux propriétés : `method` et `extraArgs`.

Penchons-nous donc sur les informations inhérentes au jeu de photos. Nous sommes intéressés par son titre, sa description, son nombre de photos, sa photo représentative (une des photos du jeu, mise en avant) et son propriétaire. Ce dernier point nécessitera d'ailleurs une deuxième requête, car au niveau du jeu de photos (le *photoset*), nous n'avons que l'identifiant (NSID) du propriétaire.

Voici le résultat XML que Flickr envoie pour le jeu 1603279.

Listing 9-40 La réponse REST de Flickr pour un jeu de photos

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photoset id="1603279" owner="19663157@N00" primary="73723635"
    ↗ secret="a9f7246444" server="20" photos="46">
    <title>Portraits of Mozillians</title>
    <description>Taken during lunch time on Dec. 8th, 2005
</description>
</photoset>
</rsp>
```

Que voit-on dans ce résultat ?

- L'élément racine est `photoset`.
- Les informations nécessaires à l'identification de sa photo représentative sur les serveurs de photos Flickr (`static.flickr.com`) sont les attributs `server`, `primary` et `secret`. Nous expliquerons cela en détail plus loin.
- Le nombre de photos est dans l'attribut `photos`.
- L'identifiant du propriétaire est dans l'attribut `owner`.
- Le titre et la description sont dans les éléments fils.

Cette grappe est plutôt simple. Pour revenir sur les attributs d'identification de la photo, il faut savoir que chez Flickr, une photo dispose de plusieurs versions de son fichier, en différentes tailles. Ces versions sont mises à disposition sur toute une ferme de serveurs de ressources statiques, accessibles *via* `static.flickr.com`. L'URL d'une photo est composée comme suit :

```
| http://static.flickr.com/[serveur]/[id]_[secret]_[taille].[format]
```

L'ID d'une photo correspond ici à l'attribut `primary` du `photoset`. La partie `secret` est là pour obliger les systèmes clients à requérir l'API afin d'obtenir les informations nécessaires, diminuant ainsi considérablement les risques de saturation, d'utilisation non autorisée, et d'attaque en déni de service.

La taille peut être l'une des valeurs suivantes :

- `_s (square)` pour la vignette carrée (destinée à un affichage en masse, comme dans la page d'un jeu de photos, ou dans notre propre page pour ce chapitre) ;
- `_t (thumbnail)` pour la vignette ;
- `_m (medium)` pour la petite taille (environ 50 %) ;
- rien pour la taille moyenne, destinée à l'affichage par défaut ;
- `_o (original)` pour la photo originale.

Une photo n'a pas forcément toutes ces variantes disponibles. On peut demander la liste des tailles d'une photo (dimensions, URL, etc.) avec la méthode flickr.photos.getSizes.

Enfin, le format de toutes les versions dérivées est jpg. La version originale a un format précis, qu'on obtient en demandant les informations pour la photo en question.

Pour notre en-tête de jeu de photos, nous utiliserons la petite taille de la photo représentative. Voici la feuille xsl/photoset.xsl, qui transforme cette grappe en ce qui nous intéresse.

Listing 9-41 La feuille XSLT xsl/photoset.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/photoset"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="photoset">
        <div id="photoset">
            <xsl:element name="img">
                <xsl:attribute name="id">
                    <xsl:value-of select="@id"/>
                </xsl:attribute>
                <xsl:attribute name="src"> ①
                    http://static.flickr.com/<xsl:value-of
                        ↪ select="@server"/>/<xsl:value-of select="@primary"/>
                        ↪ _<xsl:value-of select="@secret"/>_m.jpg
                </xsl:attribute> ②
            </xsl:element>
            <h1><xsl:value-of select="title"/></h1>
            <p class="description"><xsl:value-of select="description"/></p>
            <p class="photoCount">
                <xsl:value-of select="@photos"/>
                photo<xsl:if test="@photos > 1">s</xsl:if> ③
                de
                <span id="photoset-owner"></span> ④
            </p>
        </div>
    </xsl:template>
    <xsl:template match="err">
        <p class="error">
            Erreur n° <xsl:value-of select="@code"/>&nbsp;:
            <xsl:value-of select="@msg"/>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

Là aussi, nous prenons la peine de fournir une représentation en cas d'erreur. Notez la composition de l'attribut `src` pour l'image, de la ligne ① à la ligne ② : elle suit notre schéma. Nous sommes obligés de « coller » les composants pour éviter les espaces dans l'URL, qui pourraient troubler Flickr.

Remarquez aussi qu'en ③, nous avons pris la peine de gérer le pluriel. Quant au `span` en ④, il est vide car nous n'avons pas encore le nom, ni l'URL de profil, du propriétaire : pour les avoir, il nous faudra une deuxième requête, que nous ferons plus loin. En mettant un ID au `span`, nous pourrons mettre à jour son contenu plus tard.

Avant d'aller plus loin, pensez à réaliser un squelette XSLT valide pour les trois autres fichiers XSL référencés par `gXSLT.sheets` : sans eux, nous aurons une erreur de traitement, et notre fonction de rappel, qui doit requérir les informations du jeu de photos, ne sera jamais appelée.

Ceci étant assuré, nous allons pouvoir réaliser la fonction `getPhotoset`, qui va chercher un jeu de photos dont on lui donne l'identifiant.

Listing 9-42 Notre fonction `getPhotoset`

```
function getPhotoset(id) {
    $('#results').update('');
    var url = FLICKR_TEMPLATE.evaluate({ ①
        method: 'photosets.getInfo',
        extraArgs: '&photoset_id=' + id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var ctx = new ExprContext(data); ②
            var ownerNSID = xpathEval('/rsp/photoset/@owner',
                → ctx).stringValue();
            var html = xsltProcess(data, gXSLT.photoset); ③
            $('#results').update(html);
            // Ici, il faudra aller chercher plus d'infos...
        }
    });
} // getPhotoset
```

Cette fonction n'a rien de révolutionnaire. Remarquez tout de même en ① le mécanisme que nous utiliserons à chaque fois pour construire un appel REST à Flickr, et en ②, l'extraction manuelle du NSID de l'utilisateur Flickr propriétaire du jeu de photos. Nous allons en avoir besoin, là où se trouve pour l'instant un commentaire, pour aller chercher les informations complémentaires sur le propriétaire, afin de les

afficher dans `photoset-owner`. Remarquez enfin la transformation, qui va chercher ③ la feuille dans notre propriété exposée `gXSLT.photoset`.

Notre fonction étant prête, il ne nous reste plus qu'à compléter l'initialisation de notre page.

Listing 9-43 Une initialisation un peu plus complète

```
function initPage() {
    gXSLT.load(getPhotoset.curry(PHOTOSET_ID));
    new Draggable('photo');
} // initPage
```

Voilà : on charge les feuilles XSLT (c'est pourquoi je vous ai demandé de réaliser des squelettes valides pour les trois autres), et une fois qu'elles sont toutes chargées et interprétées avec succès, on appelle `getPhotoset` avec notre constante `PHOTOSET_ID` comme argument pré-rempli.

Voyons déjà où cela nous mène (figure 9-12).

Figure 9-12

La récupération
des premières informations



Portraits of Mozillians

Taken during lunch time on Dec. 8th, 2005

46 photos de

Terminé YSlow 1.271s

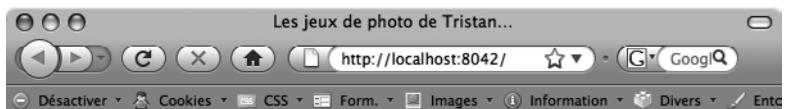
Cela manque de cachet, tout de même. Sans doute pouvons-nous améliorer les choses en ajoutant quelques règles CSS.

Listing 9-44 Ajouts dans client.css pour l'en-tête du jeu de photos

```
#photoset h1 {  
    font-family: "Bistream Vera Serif", serif;  
    color: #444;  
    font-size: 150%;  
}  
  
#photoset .description {  
    color: #777;  
}  
#photoset img {  
    float: left;  
    margin: 0 1em 1em 0;  
}  
  
#photoset .photoCount {  
    font-size: smaller;  
    color: #444;  
}
```

Notez que l'image flotte à gauche. Il faudra que l'élément suivant, que nous ajoutons tout à l'heure, soit stylé en `clear: both`. Voyons le nouveau look de notre en-tête (figure 9-13).

Figure 9-13
L'en-tête avec des règles CSS



Portraits of Mozillians

Taken during lunch time on Dec. 8th, 2005

46 photos de

Terminé YSlow 1.07s

Voilà qui n'est déjà pas mal ! Évidemment, il manque quelque chose après le « 46 photos de ». Pour cela, il nous faut simplement une nouvelle requête, afin

d'obtenir des informations sur l'utilisateur. La méthode `flickr.people.getInfo` renvoie un résultat du type suivant.

Listing 9-45 Résultat XML d'une requête d'informations utilisateur

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<person id="19663157@N00" nsid="19663157@N00" isadmin="0"
    ➔ ispro="1" iconserver="13">
    <username>nitot</username>
    <realname>Tristan Nitot</realname>
    <mbox_sha1sum>4f24c004c2d5c0cfced422c6afe26d806616919
    </mbox_sha1sum>
    <location>Paris, France</location>
    <photosurl>http://www.flickr.com/photos/nitot/</photosurl>
    <profileurl>http://www.flickr.com/people/nitot/</profileurl>
    <mobileurl>http://www.flickr.com/mob/photostream.gne?id=627752
    </mobileurl>
    <photos>
        <firstdatetaken>2005-01-12 16:37:44</firstdatetaken>
        <firstdate>1115996869</firstdate>
        <count>469</count>
    </photos>
</person>
</rsp>
```

Remarquez que Tristan est prolifique (près de 500 photos). Nous sommes particulièrement intéressés par son nom complet et l'URL de son profil utilisateur, sur laquelle nous comptons lier. Là aussi, une petite feuille XSLT, `photoset_owner.xsl`, va s'occuper de tout.

Listing 9-46 La feuille xsl/photoset_owner.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ➔ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/person"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="person">
        <xsl:element name="a">
            <xsl:attribute name="href">
                <xsl:value-of select="profileurl"/>
            </xsl:attribute>
            <xsl:value-of select="realname"/>
        </xsl:element>
    </xsl:template>
```

```

</xsl:template>
<xsl:template match="err">
    <span class="error">❶
        Erreur n&deg; <xsl:value-of select="@code"/>&nbsp;:
        <xsl:value-of select="@msg"/>
    </span>
</xsl:template>
</xsl:stylesheet>

```

C'est sans doute la feuille XSLT la plus simple du chapitre. Nous gérons tout de même les erreurs, mais comme vous pouvez le remarquer en ❶, nous utilisons ici un `span` au lieu d'un `p`. En effet, le résultat de cette feuille est censé apparaître dans l'élément `photoset-owner`, qui est lui-même un `span`. Et vous savez certainement que XHTML interdit les éléments de type bloc dans des éléments de type en ligne (ce qui est compréhensible).

Il nous reste à charger cette information juste après avoir inséré le fragment XHTML du jeu de photos dans notre DOM global. Faisons d'abord une fonction dédiée.

Listing 9-47 La fonction `getOwnerInfo`

```

function getOwnerInfo(nsid) {
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'people.getInfo',
        extraArgs: '&user_id=' + nsid
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.owner);
            $('#photoset-owner').update(html);
        }
    });
} // getOwnerInfo

```

Et ajoutons un appel après insertion du fragment conteneur, dans `getPhotoset`.

Listing 9-48 Insertion de l'appel supplémentaire dans `getPhotoset`

```

function getPhotoset(id) {
    ...
    onSuccess: function(requester) {
        ...

```

```

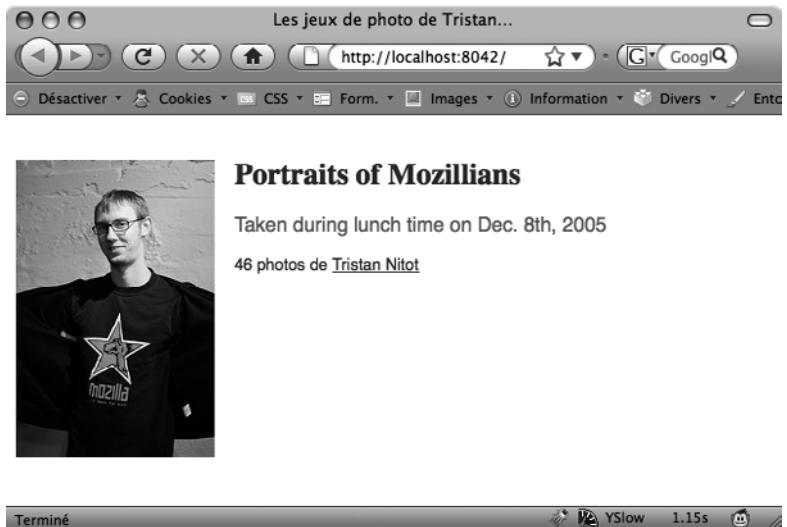
var ownerNSID = xpathEval('/rsp/photoset/@owner',
                           ➤ ctx).stringValue();
...
getOwnerInfo(ownerNSID);
}
});
} // getPhotoset

```

Observons le résultat (figure 9-14).

Figure 9-14

Les informations complètes pour le jeu de photos



Remarquez, dans la barre d'état, l'URL cible du lien : c'est bien l'URL de la page de profil de l'utilisateur.

Récupérer les photos du jeu

À présent, il nous faut les photos du jeu de photos. Pour cela, nous allons utiliser la méthode `flickr.photosets.getPhotos`. Voici un fragment du résultat.

Listing 9-49 Fragment de résultat XML pour `flickr.photosets.getPhotos`

```

<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photoset id="1603279" primary="73723635">
    <photo id="73723882" secret="015c203114" server="20"
          ➤ title="Peter Van der Beken, aka peterv" isprimary="0" />

```

```
<photo id="73723841" secret="1fde5f4ce5" server="35"
    ↪ title="Brendan Eich" isprimary="0" />
...
</photoset>
</rsp>
```

L'objectif est de produire un tableau de vignettes carrées, avec le titre de chaque photo dans les attributs `alt` et `title` des images. Le tableau ferait 10 vignettes de large.

Réaliser ce partitionnement en XSLT est vite atroce, aussi nous allons opter pour une astuce toute simple : nous allons générer une unique ligne avec toutes les vignettes, et utiliser quelques petits remplacements et une ou deux expressions rationnelles simples, pour transformer cela en série de lignes (`tr`).

Voici déjà la feuille XSLT qui va transformer le contenu XML.

Listing 9-50 La feuille xsl/photoset_photos.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/photoset"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="photoset">
        <table id="photoset-photos">
            <tbody>
                <tr>
                    <xsl:apply-templates select="photo"/>
                </tr>
            </tbody>
        </table>
    </xsl:template>
    <xsl:template match="photo">
        <td>
            <xsl:element name="img">
                <xsl:attribute name="id">
                    <xsl:value-of select="@id"/>
                </xsl:attribute>
                <xsl:attribute name="src"> ①
                    http://static.flickr.com/<xsl:value-of
                    ↪ select="@server"/>/<xsl:value-of select="@id"/>/
                    ↪ _<xsl:value-of select="@secret"/>_s.jpg
                </xsl:attribute> ②
                <xsl:attribute name="alt">
                    <xsl:value-of select="@title"/>
                </xsl:attribute>
```

```

<xsl:attribute name="title">
    <xsl:value-of select="@title"/>
</xsl:attribute>
</xsl:element>
</td>
</xsl:template>
<xsl:template match="err">
    <p class="error">
        Erreur n&deg;: <xsl:value-of select="@code"/>&nbsp;;
        <xsl:value-of select="@msg"/>
    </p>
</xsl:template>
</xsl:stylesheet>

```

Rien d'extraordinaire : on retrouve notamment, de ① à ②, notre code de composition d'URL statique pour le fichier image. Ici, nous utilisons la taille _s, pour la vignette carrée.

Commençons, comme d'habitude, par nous faire une fonction.

Listing 9-51 Notre fonction getPhotosetPhotos

```

function getPhotosetPhotos(id) {
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'photosets.getPhotos',
        extraArgs: '&photoset_id=' + id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        indicator: 'indicator',
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.photos);
            $('#results').insert(html);
        }
    });
} // getPhotosetPhotos

```

La clef ici, c'est qu'on ne remplace pas le contenu de `results` : mais qu'on ajoute un nouveau contenu en dessous de l'existant. Notre table `photoset-photos` apparaît donc sous `photoset`.

Il nous reste à appeler cette nouvelle fonction après avoir constitué l'en-tête. Pour laisser à celui-ci le temps de s'afficher d'abord, nous décalerons l'appel de 100 ms.

Listing 9-52 Modification de getPhotoset pour charger les photos

```
function getPhotoset(id) {  
    ...  
    onSuccess: function(requester) {  
        ...  
        $('results').update(html);  
        getOwnerInfo(ownerNSID);  
        window.setTimeout('getPhotosetPhotos(' + id + ')', 100);  
    }  
};  
} // getPhotoset
```

Sans les styles et sur une seule ligne, ce n'est pas terrible, mais on sent qu'on est sur la bonne voie (figure 9-15).

Figure 9-15

La table sans les styles,
en une seule ligne



Commençons par répartir cela en lignes de 10 vignettes. Pour cela, utilisons un algorithme tout simple :

- 1 Partons d'un `<tr>...</tr>` contenant toute une série de blocs `<td>...</td>`.
- 2 Encadrons chaque série de 10 blocs `<td>...</td>` par `<tr>` et `</tr>`.
- 3 Nous allons fatalement nous retrouver avec, au début, `<tr><tr>` : nous retirerons le doublon.

4 Sur la fin, suivant que nous avions un multiple de 10 vignettes ou non, nous trouverons soit `</tr></tr>`, soit `</tr><td>`. Dans le premier cas, nous retirerons le doublon. Dans le deuxième, nous insérerons un `<tr>` pour démarrer la dernière ligne.

5 C'est tout !

Bien sûr, dans l'intérêt de la modularité et de la lisibilité, nous allons placer ce code dans une fonction, mais cela va être court.

Listing 9-53 Notre fonction `adjustTable`

```
function adjustTable(html) {  
    html = html.replace(/(<td>(.|\n|\r)*?</td>){10}/img, ①  
        '<tr>$&</tr>'); ②  
    html = html.replace('<tr><tr>', '<tr>')  
        .replace('</tr><td>', '</tr><tr><td>')  
        .replace('</tr></tr>', '</tr>');  
    return html;  
} // adjustTable
```

L'expression rationnelle en ① est un peu compliquée. Le groupe entre parenthèses représente une cellule : de `<td>` jusqu'au premier `</td>` rencontré par la suite, sachant qu'entre les deux on peut trouver n'importe quoi, y compris le saut de ligne ou le retour chariot. Le quantificateur `{10}` donne le nombre d'occurrences de ce groupe que nous souhaitons. Enfin, les drapeaux `i`, `m` et `g` (rien à voir avec la balise `img` de HTML !) indiquent respectivement que nous sommes insensibles à la casse, que nous travaillons sur de multiples lignes, et que nous souhaitons ici un remplacement global (c'est-à-dire de toutes les occurrences de l'expression, pas seulement la première).

② La première édition utilisait une fonction de remplacement car Safari 2 ne gérait pas la syntaxe de remplacement `$&`, mais Safari 3 n'a plus cette lacune, alors autant faire court...

Il nous suffit maintenant d'ajuster la ligne de `getPhotosetPhotos` qui insérait le contenu :

```
    $('results').Insert(adjustTable(html));
```

En rafraîchissant, nous voyons se rapprocher la solution (figure 9-16).

Figure 9-16

Notre tableau à lignes multiples



Ajoutons à présent quelques règles CSS.

Listing 9-54 Ajouts à client.css pour notre grille de vignettes

```
#photoset-photos {
    clear: both;
    border-width: 0;
    border-collapse: collapse;
}
.photoset-photos tr {
    margin: 0;
    padding: 0;
}

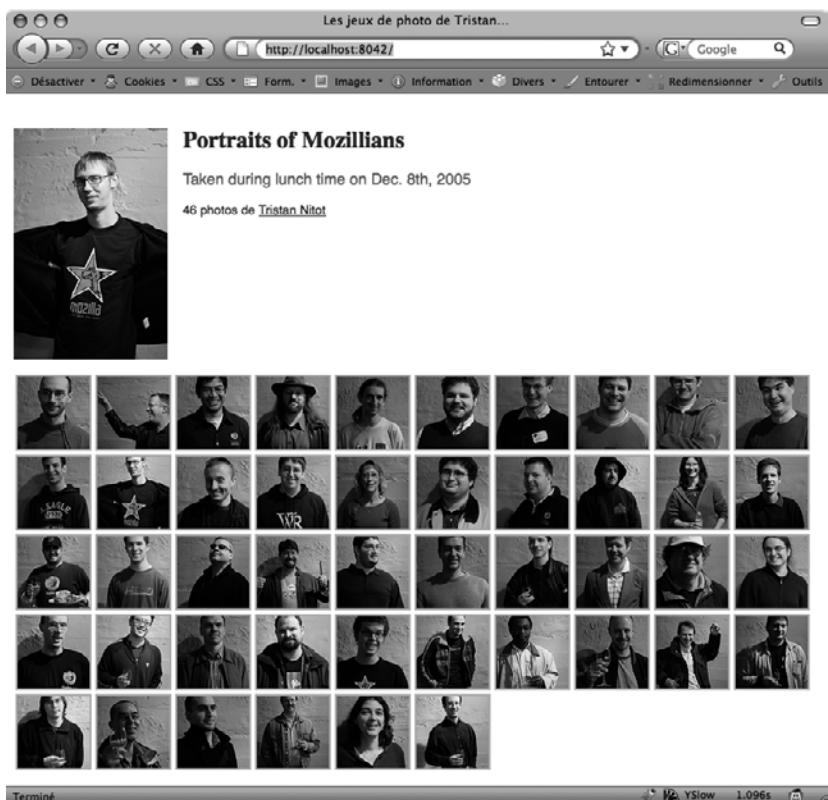
.photoset-photos td {
    padding: 0;
}

.photoset-photos img {
    border: 2px solid silver;
    margin: 0 2px;
}

.photoset-photos img:hover {
    border-color: red;
    cursor: pointer;
}
```

L'impact immédiat est dû à `clear: both` dans `#photoset-photos`, qui place la grille sous l'en-tête. Le reste vise à fournir une bordure et surligner la photo sous le curseur (figure 9-17).

Figure 9-17
Notre grille de vignettes terminée !



Afficher une photo et ses informations

Il ne nous reste plus qu'à fournir un mécanisme pour afficher une photo particulière lorsqu'on clique dessus, en taille par défaut, avec son titre, le nombre de ses commentaires si elle en a, et la liste de ses étiquettes (`tags`).

Nous avons déjà le conteneur pour cet affichage : présent depuis la première heure, notre conteneur photo n'attend que ça.

Côté événementiel, il serait stupide, voire suicidaire sur de gros volumes, de définir une inscription par vignette. Nous allons plutôt définir un gestionnaire unique au niveau de `results`. En effet, ce dernier figure toujours dans le DOM. Nous n'avons donc pas besoin de désinscrire ou réinscrire lors de rafraîchissements éventuels.

En revanche, cela impose quelques filtres : nous ne traiterons que les clics effectués sur des éléments `img` qui figurent par ailleurs dans `photoset-photos` (pour éviter notamment le clic sur la photo d'en-tête). Les autres clics seront sans effet.

On peut alors utiliser l'attribut `id` de l'image, que nous avons pris soin de définir dans `photoset_photos.xsl`. Sur la base de cet ID, on peut interroger Flickr avec la méthode `flickr.photos.getInfo` et récupérer les informations que nous souhaitons. Le résultat XML pour une requête de ce type ressemble à ceci.

Listing 9-55 Résultat XML pour une requête `flickr.photos.getInfo`

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
<photo id="73723411" secret="a4a9f2246e" server="34"
    ↪ dateuploaded="1134625094" isfavorite="0" license="1"
    ↪ rotation="0" originalformat="jpg">
    <owner nsid="19663157@N00" username="nitot"
        ↪ realname="Tristan Nitot" location="Paris, France" />
    <title>Deb Richardson, aka dria</title>
    <description />
    <visibility ispublic="1" isfriend="0" isfamily="0" />
    <dates posted="1134625094" taken="2005-12-08 21:11:28"
        ↪ takengranularity="0" lastupdate="1155766857" />
    <editability cancomment="0" canaddmeta="0" />
    <comments>2</comments>
    <notes />
    <tags>
        <tag id="627752-73723411-2889" author="19663157@N00"
            ↪ raw="Firefox">firefox</tag>
        <tag id="627752-73723411-1860001" author="19663157@N00"
            ↪ raw="firefox2005offsite">firefox2005offsite</tag>
    </tags>
    <urls>
        <url type="photopage">http://www.flickr.com/photos/nitot/73723411/
        </url>
    </urls>
</photo>
</rsp>
```

Vous voyez qu'on a tout le nécessaire, et même plus : l'auteur, qu'on connaît déjà dans notre cas, mais aussi la date de publication (en millisecondes), la date de prise de vue (format plus proche du W3DTF), celle de dernière mise à jour, les textes saisis à l'origine pour les étiquettes, etc.

Notre feuille XSLT va extraire la photo en version normale (taille par défaut), et lister le titre, la description, le nombre de commentaires (s'il y en a), et la liste des étiquettes, séparées par des virgules.

Listing 9-56 Notre feuille xsl/photo.xsl, la dernière de l'exemple

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <xsl:apply-templates select="/rsp/photo"/>
        <xsl:apply-templates select="/rsp/err"/>
    </xsl:template>
    <xsl:template match="photo">
        <h2><xsl:value-of select="title"/></h2>
        <xsl:element name="img">
            <xsl:attribute name="src">
                http://static.flickr.com/<xsl:value-of
                    ↪ select="@server"/>/<xsl:value-of select="@id"/>
                    ↪ _<xsl:value-of select="@secret"/>.jpg
            </xsl:attribute>
        </xsl:element>
        <p class="description"><xsl:value-of select="description"/></p>
        <xsl:if test="comments != 0">
            <p class="commentCount">
                <xsl:value-of select="comments"/>
                commentaire<xsl:if test="comments > 1">s</xsl:if>
            </p>
        </xsl:if>
        <xsl:if test="tags/tag">
            <p class="tags">
                &#201;tiquettes ;
                <xsl:for-each select="tags/tag">
                    <xsl:value-of select="."/>
                    <xsl:if test="position() != last()">, </xsl:if>
                </xsl:for-each>
            </p>
        </xsl:if>
    </xsl:template>
    <xsl:template match="err">
        <p class="error">
            Erreur n&deg; <xsl:value-of select="@code"/>&nbsp;;
            <xsl:value-of select="@msg"/>
        </p>
    </xsl:template>
</xsl:stylesheet>
```

Il n'y a pas de nouveautés là non plus : nous avons déjà géré des pluriels, ou séparé des éléments par des virgules (auteurs et contributeurs dans l'exemple Amazon.fr).

Il nous reste bien sûr à créer la fonction qui va chercher cela. Il s'agit d'un gestionnaire d'événement pour le clic.

Listing 9-57 Notre fonction handleThumbnailClick

```

function handleThumbClick(e) {
    $('photo').hide();
    e.stop();
    var elt = e.element(); ①
    if (elt.tagName.toLowerCase() != 'img' ②
        || !elt.childOf('photoset-photos'))
        return;
    var url = FLICKR_TEMPLATE.evaluate({
        method: 'photos.getInfo',
        extraArgs: '&photo_id=' + elt.id
    });
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(url),
        onLoading: showIndicator,
        onComplete: hideIndicator,
        onSuccess: function(requester) {
            var data = xmlParse(requester.responseText);
            var html = xsltProcess(data, gXSLT.photo);
            $('photo-contents').update(html);
            $('photo').show();
        }
    });
} // handleThumbClick

```

La fonction `Event.element` ① nous permet de récupérer l'élément réellement soumis à l'événement : comme nous allons inscrire ce gestionnaire auprès de `results`, il pourrait s'agir de n'importe quel élément dans `results`. Les lignes ② et suivantes appliquent l'algorithme de filtre que nous avons détaillé plus haut.

Inscrivons maintenant ce gestionnaire en complétant `initPage`.

Listing 9-58 Le gestionnaire est inscrit par initPage

```

function initPage() {
    gXSLT.load(function() {
        getPhotoset(PHOTOSET_ID);
    });
    $('results').observe('click', handleThumbClick);
    new Draggable('photo');
} // initPage

```

Pour que cela ait un quelconque intérêt, il faut tout de même jouer un peu avec CSS.

Listing 9-59 Ajouts à client.css pour notre visualiseur de photo

```
#photo {  
    left: 2em;  
    top: 2em;  
    background: white;  
    border: 2px solid gray;  
}  
  
#photo-closer {  
    position: absolute;  
    top: 0;  
    right: 2px;  
    font-weight: bold;  
    font-size: small;  
    color: maroon;  
    cursor: default;  
}  
  
#photo-contents {  
    margin: 1em 0.5em 0.5em 0.5em;  
}  
  
#photo h2 {  
    margin: 0 0 1em 0;  
}  
  
#photo p {  
    margin: 0.5em 0 0 0;  
}  
  
#photo .commentCount {  
    color: #444;  
}  
#photo .tags {  
    font-size: small;  
    color: gray;  
}
```

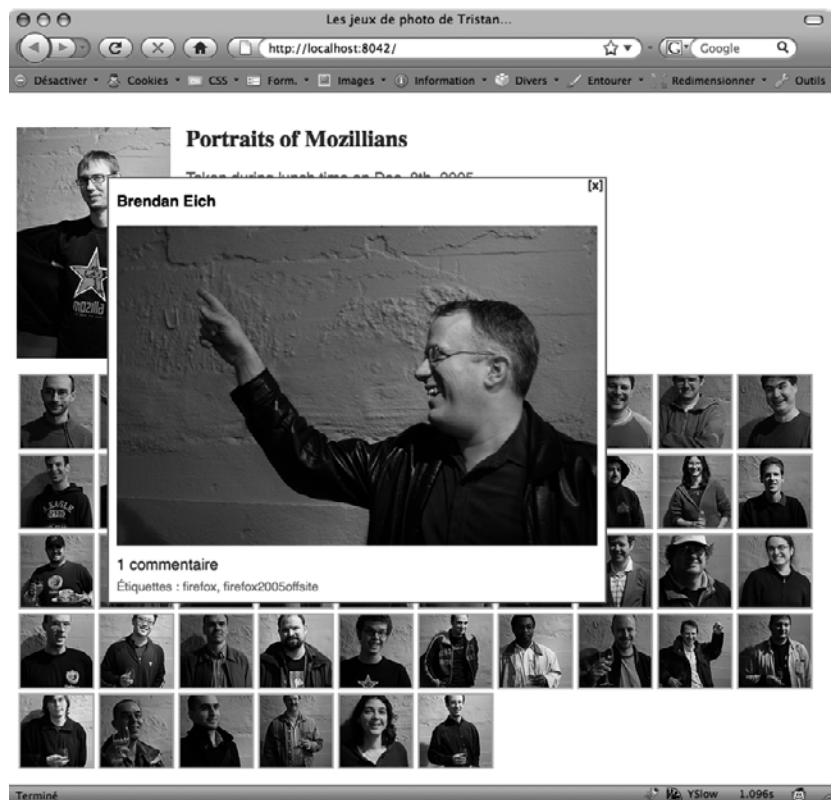
La figure 9-18 montre le résultatat d'un clic.

Ooooh !... (et au passage, je vous présente Brendan Eich, l'inventeur de JavaScript). En plus, on peut glisser-déplacer ! Si, si, rappelez-vous le new Draggable dans initPage.

Si vous cliquez ailleurs que sur le visualiseur, mais toujours dans le rectangle de results, le visualiseur est masqué (c'est pourquoi nous avons filtré uniquement *après* avoir fait un \$('photo').hide()). Ceci dit, ce serait bien que l'on puisse cliquer sur notre petite croix pour fermer, car c'est plus conforme aux attentes de l'utilisateur.

Figure 9-18

Notre visualisation de photo individuelle



Ajoutons donc ceci dans le script.

Listing 9-60 De quoi activer la croix de fermeture

```
function initPage() {
    ...
    $('#results').observe('click', handleThumbClick);
    $('#photo-closer').observe('click', closePhoto);
    new Draggable('photo');
} // initPage

function closePhoto(e) {
    e.stop();
    $('#photo').hide();
} // closePhoto
```

On rafraîchit, et voilà !

Pour aller plus loin...

Je ne vous recommanderai pas d'ouvrage particulier, mais vous trouverez ci-après quelques adresses précieuses :

- Les spécifications XPath et XSLT du W3C :
 - <http://www.w3.org/TR/xpath>
 - <http://www.w3.org/TR/xslt>
- Pour démarrer, les didacticiels de W3 Schools sont décents :
 - <http://www.w3schools.com>xpath/>
 - <http://www.w3schools.com/xsl/>
- Les portails des API vues dans ce chapitre :
 - <http://aws-portal.amazon.com/>
 - <http://www.weather.com/services/xmloap.html>
 - <http://www.flickr.com/services/api/>
- Quelques autres API REST de premier plan :
 - Google : <http://code.google.com/apis.html>
 - Yahoo! : <http://developer.yahoo.com/>
 - eBay : <http://developer.ebay.com/developercenter/rest>

10

L'information à la carte : flux RSS et Atom

Le Web 2.0, c'est aussi la mutualisation des contenus, l'échange standardisé d'informations entre sites, au travers de formats de flux. Les deux principaux formats à ce jour sont RSS et Atom. Les deux sont très employés, aussi analyserons-nous les deux, à titre d'exemple, au travers d'un service de brèves et d'un blog technique très populaire.

Techniquement, ce chapitre ne diffère guère du précédent : vous retrouverez nos requêtes Ajax en GET récupérant un contenu XML, et des transformations XSLT côté client. Vous retrouverez aussi, nécessairement, notre couche serveur dans le rôle de l'intermédiaire permettant d'éviter les problématiques de sécurité sur le navigateur. Cependant, l'utilisation que nous en ferons est différente. Nous ne dialoguerons pas avec une API, mais nous récupérerons un document régulièrement mis à jour.

Aperçu des formats

Voyons d'abord rapidement d'où viennent les formats RSS et Atom, avant de lister les informations qu'ils fournissent généralement. Nous terminerons cette introduction avec un point sur la délicate question de l'incorporation de contenu HTML dans le flux.

Une histoire mouvementée

RSS et Atom ont deux histoires très différentes, quoique liées.

RSS 0.9x et 2.0 : les « bébés » de Dave Winer

RSS est le premier format populaire de syndication. Mis au point par Dave Winer, sur la base de travaux originaux chez Netscape, pour des services comme Radio Userland, le format *Really Simple Syndication* a connu plusieurs versions successives : 0.91, 0.92, 0.93, 0.94 et finalement 2.0 (nous reviendrons dans un instant sur la mystérieuse absence de 1.0 dans cette liste). Tous ces formats partagent les mêmes caractéristiques :

- Ils sont « spécifiés » au moyen d'un vague document en ligne, illustré par quelques exemples, le tout étant très insuffisant pour permettre des implémentations fiables et interopérables.
- Ils sont le travail exclusif de Dave Winer, sans concertation aucune avec la communauté qui grandissait autour de ces formats.
- Enfin, ils sont entièrement dédiés à une utilisation de type blog, ce qui entraîne de sévères limitations pour une utilisation plus large. RSS 2.0 dispose toutefois d'un mécanisme de modules, qui permet l'extension du noyau d'éléments pour des utilisations spécialisées.

La page officielle de RSS 2.0 est <http://blogs.law.harvard.edu/tech/rss>.

RSS 1.0 : une approche radicalement différente

La version 1.0 est radicalement différente et élaborée par un groupe de travail officiel n'incluant pas Dave Winer. Ici, RSS signifie *RDF Site Summary*. Les différences sont nombreuses :

- RSS 1.0 est basé sur RDF (*Resource Description Framework*), un langage à balises au cœur des travaux du Web sémantique, qui est assez verbeux et quelque peu redondant, mais offre une structure uniforme pour tous types de données et de relations entre les données.

- RSS 1.0 est conçu pour être extensible, avec un noyau précisément défini et le reste des fonctionnalités réparti dans des modules. Trois modules sont « officiels » : le très vaste module Dublin Core (métadonnées), le module Syndication (fréquence des mises à jour) et le module Content (gestion de contenus incorporés quelconques). On compte plus d'une vingtaine de modules proposés.
- RSS 1.0 est clairement spécifié et ne laisse pas de parts d'ombre.

C'est un bon format, mais il est un peu trop lourd pour son utilisation concrète et engendre trop de complexité pour les cas usuels. En revanche, il est fiable, au sens où il permet de systématiquement lever l'ambiguïté dans ses contenus.

La page officielle de RSS 1.0 est <http://web.resource.org/rss/1.0/>.

Atom, le fruit de la maturité

Après l'apparition de RSS 1.0, Dave Winer, furieux, a immédiatement renommé sa version 0.94 en RSS 2.0, pour « conserver l'avantage », et a déclaré RSS comme « gelé » : le format était définitif, et toute extension devrait se faire au travers de modules ; par ailleurs, on ne pouvait pas utiliser le terme « RSS » pour désigner un autre format à l'avenir. En somme, il a fait l'enfant.

Le problème, c'est que RSS était très populaire et pourtant très problématique. De très nombreux cas de figure n'étaient pas encodables de façon satisfaisante dans le format : on aboutissait toujours au mieux à du contenu ambigu, au pire à des impasses.

Un groupe de travail s'est donc créé, qui a rapidement trouvé un statut officiel au sein de l'IETF, l'organisme responsable de la plupart des standards et formats qui font vivre Internet. L'objectif était de fournir un format qui soit :

- simple ;
- clair ;
- totalement spécifié (aucune zone d'ombre) ;
- sans ambiguïté aucune dans les contenus produits ;
- au moins équivalent à RSS 2.0 et RSS 1.0 en fonctionnalités ;
- capable d'incorporer facilement n'importe quel contenu, binaire, textuel balisé ou textuel quelconque ;
- un standard officiel.

En décembre 2005, sous la forme de la très officielle RFC 4287, ce format a vu le jour, avec en bonus un protocole REST de publication de contenu baptisé *Atom Publishing Protocol*.

La page officielle du format est <http://tools.ietf.org/html/rfc4287>.

Informations génériques

Tous ces formats de flux fournissent toujours les mêmes informations, peu ou prou, avec plus ou moins de détails, de fiabilité ou d'interopérabilité. On trouve d'abord deux grands niveaux :

- le flux lui-même (le *channel* RSS ou le *feed* Atom) ;
- les entrées individuelles du flux (les *items* RSS ou les *entries* Atom).

D'un flux à l'autre, les entrées individuelles peuvent contenir la totalité de l'information, ou simplement un abrégé, voire une simple référence à la ressource complète sur le Web.

Pour le flux lui-même, on dispose généralement des informations suivantes :

- son titre ;
- une URL de ressource correspondante sur le Web ;
- sa description sommaire ;
- sa langue ;
- ses dates de première parution et de dernière mise à jour ;
- ses informations légales (auteur, copyright, etc.) ;
- une identité visuelle éventuelle (par exemple le logo de l'éditeur).

Pour une entrée individuelle, on trouve souvent :

- son titre ;
- une URL vers la ressource complète ;
- un identifiant unique ;
- tout ou partie de son contenu ;
- ses dates de première parution et de dernière mise à jour ;
- des informations de catégorie ;
- son ou ses auteur(s).

Le casse-tête du contenu HTML

Tous ces formats de flux reposent au final sur XML. Il s'agit de documents balisés, qui se doivent de constituer des documents XML correctement formés. Ils répondent à une grammaire, théoriquement formalisée dans un document dédié (DTD, schéma XML ou Relax NG) ; toutefois, RSS 2.0 n'a pas de grammaire formalisée.

Les contenus HTML et XHTML utilisent eux aussi des balises. En HTML, le balisage ne constitue pas forcément du XML correctement formé : on peut se passer de fermer des balises, d'encadrer des valeurs d'attributs par des guillemets, d'utiliser

une casse spécifique... On peut même entrelacer les éléments plutôt que de respecter une imbrication cohérente ! Un tel contenu, utilisé tel quel, « casserait » donc le flux.

Quand bien même le contenu est du XHTML valide, sa grammaire ne fait pas partie de celle du format de flux dans lequel on souhaite l'incorporer. Pour permettre aux analyseurs XML de s'y retrouver en lisant le flux, il faut avoir recours soit à l'encodage (typiquement, RSS 2.0), soit aux espaces de noms (solution 100 % XML employée par RSS 1.0 et Atom), ce qui exige toutefois un traitement plus perfectionné côté client.

Au final, RSS 2.0 trouve ici ses limites, avec un problème dit du « double encodage ». Sans entrer dans les détails, on peut tomber sur des cas où, en analysant un contenu d'élément `description`, on n'est pas en mesure de déterminer avec certitude si un fragment doit être « décodé » ou non.

Atom et RSS 1.0 n'ont pas ce problème. Atom en particulier, au travers de son élément `atom:content`, permet l'encapsulation facile de n'importe quel type de contenu : textuel, XHTML, et même binaire.

Passons maintenant à la pratique.

Récupérer et afficher un flux RSS 2.0

Pour illustrer l'utilisation d'un flux RSS 2.0 (qui est de loin la variante RSS la plus employée), nous allons consulter les brèves « Client Web » publiées par le Journal du Net Développeurs, publication en ligne du Benchmark Group.

Ce flux contient des informations succinctes (ce qu'on appelle classiquement des « brèves ») centrées sur l'univers des technologies web côté client (typiquement le sujet de ce livre). L'adresse du flux RSS 2.0 pour ces brèves est <http://developpeur.journaldunet.com/rss/breve/client-web/>. Si vous vous y rendez, vous tombez pourtant sur un document XHTML tout ce qu'il y a de plus banal. Où donc se trouve le flux ?

Devant vous. Vous êtes en fait en train de consulter un document XML doté d'une feuille XSLT appliquée directement par votre navigateur. Nous utiliserons quant à nous notre propre feuille XSLT et reformaterons d'ailleurs les informations de dates et les titres.

Format du flux

En examinant le flux RSS en question, vous découvrez à peu près ceci (j'ai réindenté, abrégé des parties et retiré des fragments, dans un objectif global de lisibilité de l'extrait).

Listing 10-1 Fragments du flux RSS du JDN Développeurs

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?xmlstylesheet type="text/xsl"
  href="http://developpeur.journaldunet.com/rss/html_include/style/rss.xsl"?>
<rss version="2.0"> ①
  <channel> ②
    <title>Les br&#232;ves du Journal du Net...</title> ③
    <link>http://developpeur.journaldunet.com/</link>
    <description>Les br&#232;ves du Journal du...</description>
    <language>fr</language>
    <pubDate>Mon, 04 Sep 2006 17:23:09 +0200</pubDate>
    <lastBuildDate>Fri, 08 Sep 2006 19:00:26 +0200</lastBuildDate> ④
    <copyright>&#xA9; Benchmark Group</copyright>
    <image> ⑤
      <title>Les br&#232;ves du Journal du Net...</title>
      <link>http://developpeur.journaldunet.com/</link>
      <description>Les br&#232;ves du Journal du...</description>
      <url>http://developpeur.j.../logo_jdn_developpeurs.gif</url>
    </image>
    <item> ⑥
      <title>Client Web &#62; Firefox 2 en beta 2</title>
      <link>http://...ent-web/4614/firefox-2-en-beta-2.shtml</link>
      <guid>http://...ent-web/4614/firefox-2-en-beta-2.shtml</guid>
      <description>La prochaine &#233;volution du navigateur Open
        &#224; Source se dessine &#224; l'horizon : outre un rafra&#238;,
        &#224; chissement de son interface graphiq...</description> ⑦
      <pubDate>Mon, 04 Sep 2006 17:23:09 +0200</pubDate> ④
      <category domain="http://journaldunet.com/breve/client-web/">
        &#224; Client Web</category>
    </item>
    ...
  </channel>
</rss>
```

L'élément racine est `rss` ①, originellement pour laisser la porte ouverte à un emploi multicanaux, même si cela n'a jamais eu lieu. On descend donc encore d'un niveau dans `channel` ②, qui représente le flux à proprement parler.

Les éléments `title`, `link` et `description` ③ sont les trois éléments incontournables de RSS. Pour un `item`, `description` fournit soit une version abrégée du contenu, soit le contenu total. Ici, le JDN Développeurs a opté pour un contenu raccourci, sans

HTML ⑦, ce qui évite bien des problèmes. Notez toutefois que les accents sont encodés en dépit de la déclaration du jeu de caractères dans le prologue XML (première ligne), afin de se « blinder » contre les couches serveur ou client mal faites. On utilise ici les codes Unicode des caractères accentués (233 pour le « é », 224 pour le « à », 238 pour le « î », etc.).

En RSS, les dates et heures ④ sont fournies dans le bon vieux format de la RFC 822 (pas la 2822, qui lui a succédé en avril 2001 et étend les possibilités ; la 822, ce dinosaure datant... du 13 août 1982, dix ans avant le Web !). Cet ancien format a toutefois un avantage : il est facile à interpréter en JavaScript avec l'objet Date.

Après les métadonnées du flux lui-même, on trouve son identité visuelle, exprimée par un logo du JDN Développeurs ⑤ et enfin un élément item ⑥ par entrée individuelle ; ici donc, un item par brève.

Préparer le terrain

Commencez par faire un répertoire de travail rss2, dans lequel on retrouve notre bon vieux server.rb (oui, le même que depuis le début du chapitre 9), ainsi bien sûr que docroot et ses sous-répertoires Ajaxslt et xs1. Nous nommerons la feuille XSLT breves.xs1. Réduisez-la à un squelette valide.

Dans docroot, nous retrouvons bien sûr prototype.js, client.js, client.css, spinner.gif et index.html.

La page est absolument triviale : elle contient seulement l'indicateur de chargement et de mise en forme ainsi qu'un conteneur de résultats. Même pas de titre ! C'est logique : nous utiliserons dynamiquement le titre du flux. Voici index.html.

Listing 10-2 Notre page index.html, toute simple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  → xml:lang="fr-FR">
<head>
  <meta http-equiv="Content-Type" content="text/html;
    → charset=iso-8859-15" />
  <title>Les brèves Client Web du JDN Développeurs</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="Ajaxslt/misc.js"></script>
  <script type="text/javascript" src="Ajaxslt/dom.js"></script>
  <script type="text/javascript" src="Ajaxslt/xpath.js"></script>
  <script type="text/javascript" src="Ajaxslt/xslt.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
```

```
<body>
<div id="indicator" style="display: none;"></div>
<div id="results"></div>
</body>
</html>
```

Voyons à présent le début de notre feuille de styles. Nous y ajouterons le nécessaire après avoir obtenu le contenu XHTML que nous souhaitons.

Listing 10-3 Premier jet de client.css

```
body {
    font-family: sans-serif;
    font-size: 12pt;
}

#indicator {
    position: absolute;
    top: 10px; left: 10px;
    height: 16px; width: 20em;
    color: gray;
    font-size: 14px;
    line-height: 16px;
    background: url(spinner.gif) no-repeat;
    padding-left: 20px;
}

#results {
    margin-top: 34px;
}
```

Pour finir, voyons le squelette classique de `client.js`.

Listing 10-4 Squelette classique restant pour client.js

```
var xsltSheet;

function showIndicator() {
    $('#indicator').update('').show();
} // showIndicator
```

```
function hideIndicator() {
    $('#indicator').hide();
} // hideIndicator

function initPage() {
    // Prochainement, le chargement XSLT puis flux...
} // initPage

Ajax.Responders.register({ onException: function(requester, e) {
    $('#indicator').hide();
    alert(e);
}});

logging__ = false;

document.observe('dom:loaded', initPage);
```

Tout ceci doit vous sembler désormais très familier. Passons maintenant à la définition de notre transformation.

La feuille XSLT

La feuille n'est pas très compliquée. Nous allons produire un en-tête avec le logo, le titre et la date de dernière mise à jour, puis une liste des brèves, avec leur titre en lien et la description courte. Dans notre cas, une liste de définitions (éléments dl, dt et dd) est sémantiquement très appropriée.

Voici la feuille. Pour faciliter la lecture, j'ai découpé le tout en trois *templates*, qui correspondent à l'en-tête, aux brèves et au pied de liste (informations de copyright).

Listing 10-5 Notre feuille xsl/breves.xsl

```
<?xml version="1.0" encoding="iso-8859-15"?>
<xsl:stylesheet version="1.0"
    ↪ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
    <xsl:apply-templates select="/rss/channel"/>
</xsl:template>
<xsl:template match="channel">
    <xsl:call-template name="header"/>
    <xsl:call-template name="body"/>
    <xsl:call-template name="footer"/>
</xsl:template>
```

```
<xsl:template name="header">
  <div id="header">
    <p>
      <xsl:element name="img">
        <xsl:attribute name="id">logo</xsl:attribute>
        <xsl:attribute name="src"><xsl:value-of
          ↪ select="image/url"/></xsl:attribute>
        <xsl:attribute name="alt"><xsl:value-of
          ↪ select="image/description"/></xsl:attribute>
      </xsl:element>
    </p>
    <h1><xsl:value-of select="title"/></h1>
    <p id="lastBuildDate">
      Derni&egrave;re mise &agrave; jour ;
      <span class="timestamp"><xsl:value-of
        ↪ select="lastBuildDate"/></span>
    </p>
  </div>
</xsl:template>
<xsl:template name="body">
  <dl>
    <xsl:for-each select="item">
      <dt>
        <xsl:element name="a">
          <xsl:attribute name="href"><xsl:value-of select="link"/>
        </xsl:attribute>
        <xsl:value-of select="title"/>
      </xsl:element>
      <span class="pubDate">
        &middot;
        <span class="timestamp"><xsl:value-of select="pubDate"/>
      </span>
    </span>
    </dt>
    <dd>
      <p><xsl:value-of select="description"/></p>
    </dd>
  </xsl:for-each>
</dl>
</xsl:template>
<xsl:template name="footer">
  <p class="footer"><xsl:value-of select="copyright"/></p>
</xsl:template>
</xsl:stylesheet>
```

J'ai souligné les éléments importants avec leurs classes et ID, pour vous permettre de bien voir les relations avec les règles CSS que nous ajouterons par la suite.

Chargement et formatage du flux

Il est temps d'ajouter le désormais traditionnel code de chargement de feuille XSLT puis de flux. N'ayant qu'une seule feuille, il est inutile de recourir à des objets dédiés comme le gXSLT de l'exemple Flickr au chapitre 9. Voici notre `initPage`.

Listing 10-6 Notre fonction `initPage` terminée

```
function initPage() {
    new Ajax.Request('xsl/breves.xsl', {
        method: 'get',
        indicator: 'indicator',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            getFeed();
        }
    });
} // initPage
```

Par souci de clarté et de modularité, nous avons délégué le chargement et le traitement du flux à une fonction à part, `getFeed`. La voici.

Listing 10-7 Notre fonction `getFeed`

```
FEED_URL = 'http://developpeur.journaldunet.com/rss/breve/client-web/';

function getFeed() {
    $('#results').update('');
    showIndicator();
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(FEED_URL),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('#indicator').update('Mise en forme');
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, xsltSheet);
                $('#results').update(html);
                hideIndicator();
            }, 10);
        }
    });
} // getFeed
```

Il n'y a là que du grand classique. Voilà, nous avons le minimum nécessaire. Lançons `serveur.rb` et naviguons jusqu'à `http://localhost:8042` (figure 10-1).

Figure 10-1
Notre flux RSS chargé



Évidemment, un peu de CSS ne fait jamais de mal.

Listing 10-8 Des règles CSS pour rendre tout ceci plus joli

```
#results h1 {
    font-family: Georgia, serif;
    font-size: 140%;
    color: #555;
}

#lastBuildDate {
    font-size: smaller;
    color: gray;
}

#results p.footer {
    font-size: small;
    color: gray;
    border-top: 1px solid silver;
    margin: 1em 0;
}

#results dl {
    margin: 1em 0 0 1em
}
```

```
#results dt a {  
    font-weight: bold;  
    color: green;  
}  
  
#results dt a:visited {  
    color: #050;  
}  
  
#results span.pubDate {  
    color: gray;  
    font-size: smaller;  
    font-weight: normal;  
}  
  
#results dd {  
    margin-left: 1em;  
}  
  
#results dd p {  
    margin: 0.3em 0 1em 0;  
    font-size: 90%;  
    color: #444;  
}
```

Rafraîchissons, nous obtenons la figure 10-2.

Figure 10–2

C'est déjà beaucoup mieux.



Il nous reste tout de même à nous occuper des dates (le format actuel n'est pas très agréable pour des francophones) et des titres (ce préfixe « Client Web » est agaçant).

Ajustements des dates et titres

Comme d'habitude, nous allons dédier une fonction `adjustData` à ces traitements et utiliser des expressions rationnelles. Si vous examinez notre feuille XSLT, vous voyez que nous avons isolé toute date produite dans un `span` de classe `timestamp`. Quant aux titres, ils sont dans des éléments `a` dont le libellé commence par « Client Web > ». Le code de traitement donne ceci.

Listing 10-9 Notre ajustement de contenus

```
RE_TITLE = '(<a .*?>)Client Web\\s*&gt;\\s*(.*?)(</a>)';
RE_TIMESTAMP = '(<span class="timestamp">)(.*?)(</span>)';

function adjustData(html) {
    var html = html.gsub(RE_TIMESTAMP, function(match) {
        match[2] = new Date(match[2]).toLocaleString();
        return match[1] + match[2] + match[3];
    });
    html = html.replace(new RegExp(RE_TITLE, 'img'), '$1$2$3');
    return html;
} // adjustData
```

Figure 10-3

Notre affichage de flux RSS prêt à l'emploi



Évidemment, il ne faut pas oublier d'ajuster `getFeed`.

Listing 10-10 Le fragment de `getFeed` utilisant l'ajustement

```
var tmr = window.setTimeout(function() {  
    ...  
    $('results').update(adjustData(html));  
    hideIndicator();  
}, 10);
```

Après un petit rafraîchissement, nous obtenons la figure 10-3.

Affichage plus avancé et flux Atom 1.0

À présent que nous avons vu un exemple simple autour de RSS 2.0, il est temps de voir un exemple plus complet, autour d'Atom. Nous utiliserons l'incontournable Standblog, le blog de Tristan Nitot, président de Mozilla Europe et fervent évangéliste des standards du Web depuis bien des années.

Le blog de Tristan tourne sur Dotclear (<http://www.dotclear.com>), l'excellent moteur de blog conforme aux standards d'Olivier Meunier. Comme la plupart des outils de blog, Dotclear fournit d'office des flux Atom et permet au blogueur de choisir s'il (ou elle) souhaite incorporer le texte complet des billets dans le flux, ou simplement un abrégé.

Tristan choisit le texte intégral, balisage XHTML compris, ce qui sert parfaitement notre exemple : cela ouvre des possibilités amusantes côté client, avec des effets script.aculo.us, pour afficher d'abord les textes abrégés, et faire apparaître sur demande, de façon un peu vivante, les contenus complets.

En revanche, cela va nous demander un peu de travail supplémentaire. En effet, Dotclear génère des flux Atom avec des contenus en mode HTML encodé (un des types possibles pour l'élément `atom:content`), ce qui signifie que le HTML est encodé dans un nœud `texte`, de façon assez similaire à RSS (mais sans l'ambiguïté, le format précisant qu'on n'utilise pas de double encodage).

Ce nœud ne peut pas être transformé en HTML par XSLT : c'est à notre JavaScript de manipuler le texte du fragment XHTML résultat de la transformation. Et cette transformation pose quelques problèmes techniques de portabilité, qui donneront parfois du code un peu « à la main ». Si Dotclear produisait des éléments `atom:content` recourant aux espaces de noms, nous n'aurions rien eu à faire. Olivier a certainement une bonne raison d'avoir procédé ainsi.

Si ce sujet (`atom:content` et ses différents modes) vous intéresse, il est traité dans tous les détails, avec des exemples, pages 133 à 142 de l'ouvrage sur RSS et Atom mentionné en fin de chapitre.

Notre flux Atom

Commençons par examiner le flux Atom du Standblog. Voici une vue fragmentée, ré-indentée, etc..

Listing 10-11 Vue filtrée du flux Atom du Standblog

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<feed xmlns="http://www.w3.org/2005/Atom" ①
      xmlns:sy="http://purl.org/rss/1.0/modules/syndication/" ②
      xml:lang="fr">
  <title>Standblog</title> ③
  <link rel="alternate" type="text/html"
        ↗ href="http://standblog.org/blog/"/>
  <link rel="self" href="http://standblog.org/dotclear/atom.php"/>
  <id>tag:standblog.org,2006:/blog/</id>
  <updated>2006-09-08T20:16:11+02:00</updated> ④
  <generator version="1.2.5" uri="http://www.dotclear.net/">DotClear
  </generator>
  <sy:updatePeriod>daily</sy:updatePeriod> ②
  <sy:updateFrequency>1</sy:updateFrequency>
  <sy:updateBase>2006-09-08T20:16:11+02:00</sy:updateBase>
  <entry xml:lang="fr"> ①
    <title>En vrac, vite fait...</title>
    <link rel="alternate" type="text/html" href="http://standblog.org/
          ↗ blog/2006/09/08/93114894-en-vrac-vite-fait" />
    <updated>2006-09-08T20:16:11+02:00</updated>
    <id>tag:standblog.org,2006-09-08:/dotclear/93114894</id>
    <author><name>Tristan</name></author>
    <category term="En-vrac" label="En vrac"/>
    <summary>...oui, encore plus vite fait que d'habitude :-(  

Michel de Guilhermier aime iGraal (merci Yann pour le lien !) ;  

Voici une offre d'emploi pour un développeur XUL ;  

The difficulty of simplicity... Je n'aurais pas dis mieux !  

Adobe laisse tomber son plug-in SVG... Certes, Opera et...  

    </summary>
    <content type="html"> &lt;p&gt;...oui, encore plus vite fait que  

d'habitude :-)&lt;/p&gt; ⑤  

&lt;ul&gt;  

&lt;li&gt;&lt;a href="http://micheldeguilhermier.typepad.com/  

mdegblblog/2006/09/brand_new_conna.html" hreflang="fr"&gt;Michel de  

Guilhermier aime iGraal&lt;/a&gt;  

(merci Yann pour le lien !)&nbsp;;&lt;/li&gt;
```

```
&lt;li&ampgtVoici une &lt;a href="http://fredericdevillamil.com/articles/2006/09/04/offre-demploi-d%C3%A9veloppeur-php-xul"; hreflang="fr"&ampgtoffre d'emploi pour un développeur XUL&lt;/a&ampgt&nbsp;&lt;/li&ampgt

&lt;li&ampgt&lt;a href="http://www.allpeers.com/blog/2006/09/07/the-difficulty-of-simplicity"; hreflang="en"&ampgtThe difficulty of simplicity&lt;/a&ampgt... Je n'aurais pas dis mieux&nbsp;!&lt;/li&ampgt
&lt;li&ampgt&lt;a href="http://georezo.net/forum/viewtopic.php?pid=56305#p56305"; hreflang="fr"&ampgtAdobe laisse tomber son plug-in SVG&lt;/a&ampgt... Certes, Opera et Firefox intègrent une partie de la spécification, mais ça n'est pas une bonne raison à mon sens. Il faut dire que le SVG est en concurrence partielle avec Flash, lequel dépend d'Adobe depuis le rachat de Macromedia... la &lt;a href="http://www.adobe.com/svg/pdfs/ASV_EOL_FAQ.pdf"; hreflang="en"&ampgtFAQ de finde vie du plug-in SVG (format PDF)&lt;/a&ampgt est très révélatrice&nbsp;&lt;/li&ampgt

&lt;li&ampgtMerci à Filip qui me signale que ma &lt;a href="#"; hreflang="fr"&ampgtphoto de la tour Eiffel&lt;/a&ampgt est reprise &lt;a href="http://www.smartphone.net/smartphonethoughts/software_detail.asp?id=2520"; hreflang="en"&ampgtici&lt;/a&ampgt.. Dans la même série, elle a déjà été publiée en Nouvelle Zélande, et de nombreuses photos de votre serviteur servent d'exemples dans un &lt;a href="http://gimp4you.eu.org/livre/"; hreflang="fr"&ampgtnouveau livre sur Gimp&lt;/a&ampgt.&lt;/li&ampgt

&lt;/ul&ampgt
  </content>
</entry>
...
</feed>
```

Vous voyez en ① les deux éléments clefs du flux : `feed`, élément racine effectif, qui représente le flux (on casse donc d'entrée de jeu la compatibilité avec RSS), et `entry`, qui représente une entrée individuelle.

Les lignes marquées ② montrent un module d'extension par espaces de noms en action : le module officiel Syndication. Le préfixe `sy` est associé à l'URL officielle de l'espace de noms pour le module, et les éléments issus de cet espace emploient le préfixe. Il s'agit du mécanisme fondamental d'extension en XML. Le « X » de XML, *eXtensible*, c'est beaucoup grâce à lui.

Vous remarquez en ③ qu'on retrouve `title` et `link`, même si en Atom, `link` a un rôle beaucoup plus clair que dans RSS, tout en restant polyvalent. En revanche, plus de schizophrénie avec `description`, tantôt résumé et tantôt contenu complet : Atom clarifie les rôles avec `summary` et `content`.

Vous observez en ④ que les dates/heures en Atom utilisent un format récent, de plus en plus répandu : le W3DTF (*W3C Date/Time Format*). C'est très bien, mais l'objet JavaScript `Date` n'a pas encore suivi le mouvement : ne serait-ce qu'en raison de la représentation du décalage GMT, nous ne pourrons nous contenter d'une expression rationnelle pour reformater et interpréter les dates : il faudra les décoder manuellement.

Enfin, remarquez en ⑤ l'élément `content` et son attribut `type`, qui vaut ici `html`. Dans ce mode, le contenu est encodé : le client *sait* qu'il doit décoder ce contenu une et une seule fois (c'est-à-dire que si le décodage produit de nouvelles entités, par exemple avec `>` qui donne `>`, on en reste là).

Tristan étant parfois prolifique, on se retrouve de temps en temps avec de grosses grappes de HTML encodé à transformer, avec JavaScript, en HTML normal. En raison de la taille globale du HTML obtenu par XSLT (qui représente tout le flux), nous aurions quelques problèmes avec les expressions rationnelles sur certains navigateurs ; ainsi, Konqueror 3.5.2 « planterait » purement et simplement, tandis que Firefox se bornerait à 4 Ko de texte pour certaines opérations relatives au décodage des entités HTML... Il s'agit donc de tâches que nous devrons à nouveau effectuer manuellement...

Préparation de notre lecteur de flux

Allez, c'est parti. Vous avez dû tout retenir maintenant, mais nous vous redonnons tout de même les étapes habituelles : faites un répertoire de travail `atom`, dans lequel vous installez notre vaillant `serveur.rb` (toujours le même), le répertoire `docroot` et ses sous-répertoires `Ajaxslt` et `xsl`. La feuille XSLT sera `standblog.xsl`. Réduisez-la à un squelette valide.

Dans `docroot`, vous placez bien sûr `prototype.js`, `client.js`, `client.css`, `spinner.gif` et `index.html`, comme toujours.

La page est toujours aussi triviale : on a juste l'indicateur de chargement et de mise en forme, et un conteneur de résultats. Au titre près, c'est exactement celle de l'exemple précédent, avec tout de même un chargement `script.aculo.us` en plus, en prévision d'effets à venir (par conséquent, ajoutez `scriptaculous.js` et `effects.js` dans `docroot`). Voici le HTML.

Listing 10-12 Notre page `index.html`, à peine différente de la précédente

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR"
  xml:lang="fr-FR">
```

```
<head>
  <meta http-equiv="Content-Type" content="text/html;
    ↪ charset=iso-8859-15" />
  <title>Le Standblog</title>
  <link rel="stylesheet" type="text/css" href="client.css" />
  <script type="text/javascript" src="prototype.js"></script>
  <script type="text/javascript" src="scriptaculous.js?load=effects">
  </script>
  <script type="text/javascript" src="Ajaxslt/misc.js"></script>
  <script type="text/javascript" src="Ajaxslt/dom.js"></script>
  <script type="text/javascript" src="Ajaxslt/xpath.js"></script>
  <script type="text/javascript" src="Ajaxslt/xslt.js"></script>
  <script type="text/javascript" src="client.js"></script>
</head>
<body>

  <div id="indicator" style="display: none;"></div>

  <div id="results"></div>

</body>
</html>
```

Le début de notre feuille de styles est identique à l'exemple précédent. Nous y ajouterons le nécessaire après avoir obtenu le contenu XHTML que nous souhaitons, lequel différera tout de même beaucoup.

Listing 10-13 Premier jet de client.css

```
body {
  font-family: sans-serif;
  font-size: 12pt;
}

#indicator {
  position: absolute;
  top: 10px; left: 10px;
  height: 16px; width: 20em;
  color: gray;
  font-size: 14px;
  line-height: 16px;
  background: url(spinner.gif) no-repeat;
  padding-left: 20px;
}

#results {
  margin-top: 34px;
}
```

Côté script, nous pouvons conserver la base de l'exemple précédent, notamment la constante de l'URL du flux (mais évidemment, ce n'est plus la même URL), et l'expression rationnelle isolant les dates, en plus du code habituel.

Listing 10-14 Squelette classique restant pour client.js

```
FEED_URL = 'http://standblog.org/dotclear/atom.php';

RE_TIMESTAMP = '(<span class="timestamp">) (.*)(</span>)';

var xsltSheet;

function showIndicator() {
    $('indicator').update('').show();
} // showIndicator

function hideIndicator() {
    $('indicator').hide();
} // hideIndicator

function initPage() {
    // Prochainement, le chargement XSLT puis flux...
} // initPage

Ajax.Responders.register({ onException: function(requester, e) {
    $('indicator').hide();
    alert(e);
}});

logging__ = false;

document.observe('dom:loaded', initPage);
```

Passons maintenant à la définition de notre feuille XSLT.

La feuille XSLT et le formatage

Nous avons conservé le principe des trois *templates* d'en-tête, de corps et de pied. Plus de logo ici, mais toujours des entrées individuelles dans une liste de définitions (`dl`, `dt`, `dd`). En revanche, nous avons deux blocs de texte par entrée :

- un paragraphe de résumé visible, avec sur la fin du texte un lien d'affichage du contenu complet ;
- et un conteneur initialement masqué pour ce fameux contenu.

Nous ajoutons aussi à chaque entrée sa langue, fournie dans l'attribut standard `xml:lang` des éléments `entry`. C'est important sur le Standblog, car les billets sont aussi souvent en anglais qu'en français.

Voici la feuille...

Listing 10-15 Notre feuille xsl/standblog.xsl

```
<xsl:template match="/">
  <xsl:apply-templates select="/feed"/>
</xsl:template>
<xsl:template match="feed">
  <xsl:call-template name="header"/>
  <xsl:call-template name="body"/>
  <xsl:call-template name="footer"/>
</xsl:template>
<xsl:template name="header">
  <div id="header">
    <h1><xsl:value-of select="title"/></h1>
    <p id="lastBuildDate">
      Derni&egrave;re mise &agrave; jour :
      <span class="timestamp"><xsl:value-of select="updated"/>
      </span>
    </p>
  </div>
</xsl:template>
<xsl:template name="body">
  <dl>
    <xsl:for-each select="entry">
      <xsl:element name="dt">
        <xsl:copy-of select="@xml:lang"/> ①
        <xsl:element name="a">
          <xsl:attribute name="href">
            <xsl:value-of select="link[@type='text/html']/@href"/> ②
          </xsl:attribute>
          <xsl:value-of select="title"/>
        </xsl:element>
        <span class="pubDate">
          &middot;
          <span class="timestamp"><xsl:value-of select="updated"/>
          </span>
        </span>
      </xsl:element>
      <dd>
        <xsl:copy-of select="@xml:lang"/> ①
        <p class="summary">
          <xsl:value-of select="summary"/>
          <span class="toggler">[ ③
          <xsl:element name="a">
```

```
<xsl:attribute name="href">
    <xsl:value-of select="link[@type='text/html']/@href"/>
</xsl:attribute>
Voir tout l'article
</xsl:element>
] </span> ④
</p>
<div class="content" style="display: none">
    <div><xsl:value-of select="content"/></div>
</div>
</dd>
</xsl:for-each>
</dl>
</xsl:template>
<xsl:template name="footer">
    <p class="footer"><xsl:value-of select="copyright"/></p>
</xsl:template>
</xsl:stylesheet>
```

C'est la première fois que nous utilisons `xsl:copy-of` ① dans une feuille XSLT. Cette instruction copie un fragment XML dans notre résultat, en tant que XML toujours : ici, nous récupérons l'attribut `xml:lang` tel quel, ce qui n'a de sens que dans un `xsl:element` (et nous y sommes : respectivement `dt` et `dd`). Nous indiquons ainsi la langue des textes de l'entrée, ce qui sera notamment très utile aux logiciels lecteurs d'écran.

La ligne ② illustre l'obtention de l'URL du billet sur le Web : en Atom, les éléments `link` fournissent l'URL dans leur attribut `href`, et non ailleurs. Nous prenons bien soin de choisir un lien typé `text/html`, qui représente normalement la ressource (idéalement, il faudrait en plus tester que l'attribut `rel` vaut `alternate`) : il pourrait y en avoir d'autres (feuilles de styles CSS suggérées, relations hiérarchiques dans un ensemble de documents, etc.).

Les lignes ③ à ④ créent les liens de bascule d'affichage, en fin de chaque texte abrégé. Pour l'accessibilité, il est bon que ces liens, dont la vocation est d'afficher le texte complet, amènent naturellement (attribut `href`) sur le billet. Nous intercepterons leur déclenchement avec de l'*unobtrusive JavaScript* pour afficher le contenu déjà chargé, mais initialement masqué. C'est le même principe que pour un pop-up accessible.

Charger la feuille et le flux

Notre feuille est prête ; il nous reste à la charger, puis à récupérer le flux et appliquer la feuille.

Voici déjà `initPage`.

Listing 10-16 Notre fonction `initPage` qui charge la feuille puis le flux

```
function initPage() {
    new Ajax.Request('xsl/standblog.xsl', {
        method: 'get',
        indicator: 'indicator',
        onSuccess: function(requester) {
            xsltSheet = xmlParse(requester.responseText);
            getFeed();
        }
    });
} // initPage
```

Le code est presque identique à celui de l'exemple précédent : seul le nom de la feuille a changé. Voyons à présent `getFeed` et notre éternelle `adjustData`, qui ne fait rien pour l'instant.

Listing 10-17 Le chargement du flux et la transformation avec `getFeed`

```
function adjustData(html) {
    return html;
} // adjustData

function getFeed() {
    $('results').update('');
    showIndicator();
    new Ajax.Request('/xmlProxy', {
        method: 'get',
        parameters: 'url=' + encodeURIComponent(FEED_URL),
        onFailure: hideIndicator,
        onSuccess: function(requester) {
            $('indicator').update('Mise en forme');
            var tmr = window.setTimeout(function() {
                window.clearTimeout(tmr);
                var data = xmlParse(requester.responseText);
                var html = xsltProcess(data, xsltSheet);
                $('results').update(adjustData(html));
                hideIndicator();
            }, 10);
        }
    });
} // getFeed
```

Et voilà ! Nous pouvons lancer la couche serveur et tester (figure 10-4).

Figure 10-4

Le Standblog, un peu nu,
dans notre navigateur



Commençons par habiller notre Standblog en ajoutant quelques règles CSS.

Listing 10-18 Ajouts à client.css pour habiller le flux

```
#results h1 {
    font-family: Georgia, serif;
    font-size: 140%;
    color: #555;
}

p#lastBuildDate {
    font-size: smaller;
    color: gray;
}

#results p.footer {
    font-size: small;
    color: gray;
    border-top: 1px solid silver;
    margin: 1em 0;
}

#results dl {
    margin: 1em 0 0 1em
}
```

```
#results dt a {  
    font-weight: bold;  
    color: green;  
}  
  
#results dt a:visited {  
    color: #050;  
}  
  
#results span.pubDate {  
    color: gray;  
    font-size: smaller;  
    font-weight: normal;  
}  
  
#results dd {  
    margin-left: 1em;  
    font-size: 90%;  
    color: #444;  
}  
  
#results dd p {  
    margin: 0.3em 0 1em 0;  
}  
  
#results dd div.content {  
    margin: 0.3em 0 1em 0;  
    border: 1px solid gray;  
    background: #ddd;  
    padding: 1em;  
}
```

Le résultat est présenté sur la figure 10-5.

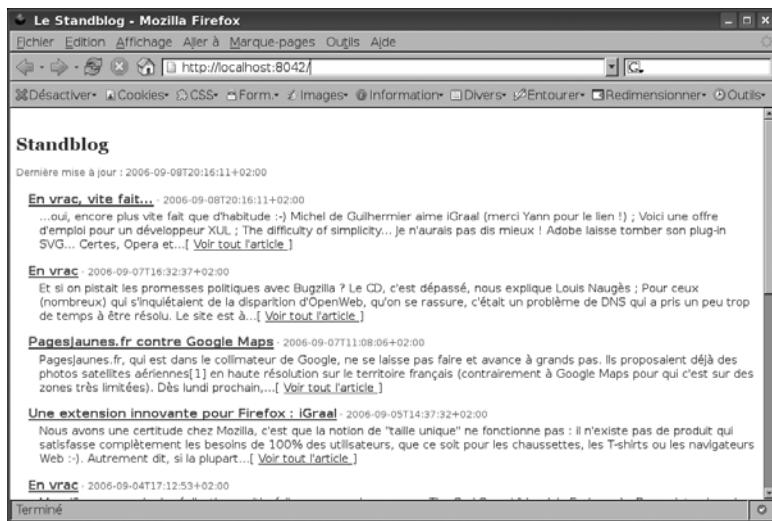
C'est déjà mieux, même si les dates W3DTF détonnent un peu. Occupons-nous maintenant de donner vie aux liens *Voir tout l'article*, qui pour l'instant sont des liens classiques.

Afficher dynamiquement les billets complets

Une fois le flux chargé, transformé et inséré dans le DOM, nous devons réagir aux clics sur les liens de bascule pour afficher le contenu complet (présent dans un *div* caché après le paragraphe abrégé) et masquer le texte abrégé.

En même temps, utilisons script.aculo.us pour rendre l'ensemble plus joli : un fondu jusqu'à disparition de l'abrégué, tandis que la version complète apparaît en défilant vers le bas, le tout synchronisé. Cela nous permettra de réviser le chapitre 8.

Figure 10–5
L'affichage initial, habillé



Définissons d'abord une fonction chargée de récupérer tous les liens de bascule et de leur affecter un gestionnaire d'événement unique. C'est l'occasion de voir \$\$ en action.

Listing 10-19 Fonction bindToggliers

```
function bindToggliers() {
    $$('#results .toggler a').invoke('observe', 'click', handleToggler);
} // bindToggliers
```

Nous allons donc écrire un gestionnaire `handleToggler`. Notez la puissance de \$\$, qui nous fournit un énumérable de tous les éléments a contenus dans un élément de classe `toggler` au sein du conteneur `results`. Imaginez juste à quoi ressemblerait cette fonction *sans* Prototype...

Voyons maintenant notre fonction `handleToggler`, qui met elle aussi Prototype lourdement à contribution.

Listing 10-20 Le gestionnaire unique `handleToggler`

```
function handleToggler(e) {
    e.stop();
    var toggler = e.findElement('p');
```

```
new Effect.Parallel([
    new Effect.BlindDown(toggler.next('div')),
    new Effect.Fade(toggler)
], { duration: 2.0 });
} // handleToggler
```

Voilà beaucoup de Prototype en très peu de lignes. Nous commençons par interrompre l'événement pour éviter la navigation normale après clic. Nous récupérons ensuite l'élément concerné (le lien qui a été cliqué), nous le passons à \$ pour obtenir un élément garanti étendu, sur lequel nous pouvons donc appeler up, ici pour remonter jusqu'au premier ancêtre de balise p.

Ensuite, nous créons une exécution parallèle de deux effets, sur une durée de deux secondes : un défilement bas du contenu complet, masqué jusqu'ici (attribut style="display: none" dans le HTML), et un fondu jusqu'à disparition du paragraphe abrégé (lien de bascule compris).

Notez comment nous désignons le contenu complet : en partant du paragraphe abrégé, et en suivant ses noeuds frères vers le bas jusqu'à tomber sur un div. Remarquez au passage que la fonction up renvoyant un élément garanti étendu dans toggler, nous pouvons appeler next directement sur celui-ci (sans encadrer par \$, par exemple).

Le HTML fourni par la feuille XSLT ajoute un div à l'intérieur de celui manipulé ici, afin de permettre l'effet SlideDown au lieu de BlindDown, qui est conceptuellement plus sympathique, mais a tendance à beaucoup clignoter sur certaines configurations. BlindDown est plus stable. Notez aussi que Fade, comme tout effet utilisant opacity, ne marche pas dans les versions actuelles de Konqueror : les utilisateurs de ce navigateur verront juste le paragraphe abrégé disparaître d'un coup, en fin d'effet.

Il ne nous reste qu'à appeler bindTogglers après insertion du HTML du flux dans le DOM de la page, au sein de getFeed.

Listing 10-21 Ajustement de getFeed pour appeler bindTogglers

```
var tmr = window.setTimeout(function() {
    ...
    $('results').update(adjustData(html));
    bindTogglers();
    hideIndicator();
}, 10);
```

Rechargeons et cliquons sur un lien de bascule (figure 10-6).

Figure 10–6

Un résultat de bascule...
surprenant.



Mais qu'est-ce que c'est que ce charabia ? Eh bien, ne vous avais-je pas dit que le HTML encodé était transmis en tant que noeud texte, et qu'il allait falloir l'interpréter nous-mêmes ? Voilà, nous y sommes. Retroussons nos manches.

Les mains dans le cambouis : interpréter le HTML encodé

Commençons par transformer ce HTML encodé en HTML décodé ; nous peaufinerons avec le formatage des dates W3DTF pour finir.

Traiter des quantités massives de HTML encodé

Le XHTML retourné par la transformation XSLT est à laisser globalement intact : seules les parties de contenu, en HTML encodé, sont à décoder. La taille totale du XHTML va bloquer les méthodes `replace` ou `gsub`, sans parler du `unescapeHTML` de Prototype, sur certains navigateurs. Nous allons donc devoir réaliser la recherche, le décodage et le remplacement « à la main ».

Voici le code dans `adjustData` qui va remplacer les fragments de HTML pour les contenus de billet par leur version décodée. Souvenez-vous qu'il y a deux `div` imbriqués dans le HTML produit : le `div` conteneur et un autre `div` simple en prévision d'un effet `SlideDown`.

Listing 10-22 La fonction adjustData avec le code de décodage ciblé

```
function adjustData(html) {  
    DIV_OPENER = '<div class="content"';  
    DIV_CLOSER = '</div>';  
    var start = 0;  
    var pos = html.indexOf(DIV_OPENER, start);  
    var result = '';  
    while (-1 != pos) {  
        var openerEnd = html.indexOf('<div>', pos) + 5;  
        result += html.substring(start, openerEnd);  
        var closerStart = html.indexOf(DIV_CLOSER, openerEnd);  
        result += safeUnescape(html.substring(openerEnd, closerStart));  
        start = closerStart;  
        pos = html.indexOf(DIV_OPENER, start);  
    }  
    result += html.substring(start);  
    return result;  
} // adjustData
```

Il s'agit de code de traitement de chaîne bête et méchant. Voyons maintenant notre fonction `safeUnescape`, qui remplace le `unescapeHTML` de Prototype afin de fonctionner même sur de gros textes quel que soit le navigateur.

Listing 10-23 La fonction safeUnescape, ou unescapeHTML garantie

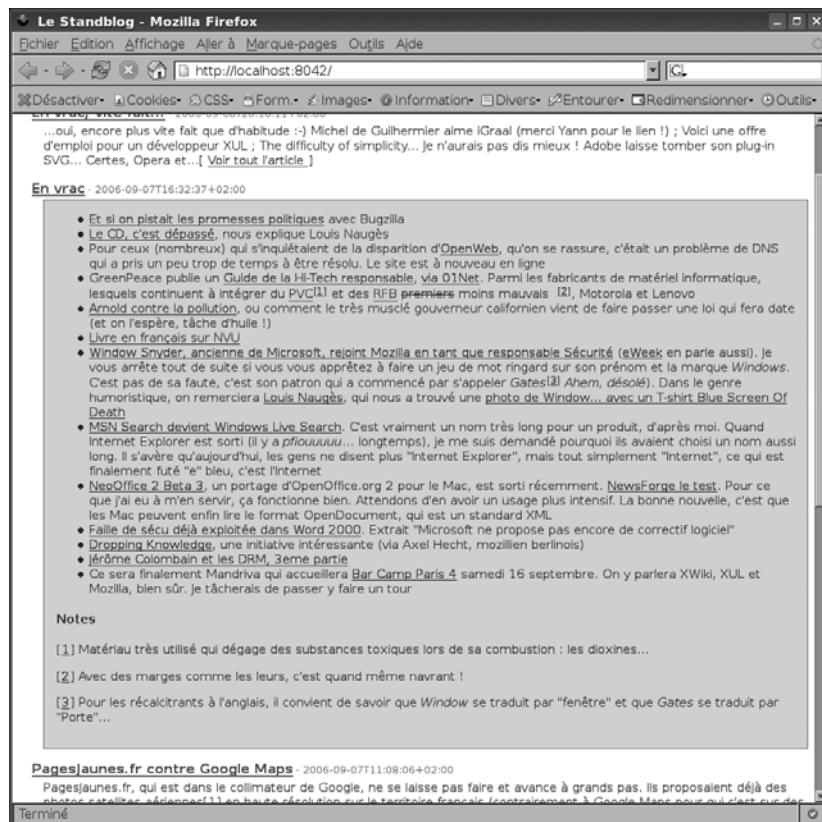
```
function safeUnescape(text) {  
    TRANSITIONS = { 'lt': '<', 'gt': '>', 'quot': '\"',  
    'apos': "\'", 'amp': '&' };  
    for (var entity in TRANSITIONS) {  
        var re = new RegExp('&' + entity + ';', 'mg');  
        text = text.replace(re, TRANSITIONS[entity]);  
    }  
    return text;  
} // safeUnescape
```

Décidément, ces objets anonymes comme *hashes* sont bien pratiques. Ici, ils nous simplifient la déclaration des associations d'entités. Nous traitons les cinq entités encodées par XML (et donc HTML) : les chevrons, les guillemets simples et doubles, et l'esperluette (&). Les drapeaux `m` et `g` pour l'expression rationnelle assurent un remplacement global sur toutes les lignes.

Le résultat est présenté à la figure 10-7.

Figure 10-7

C'est beaucoup mieux !



Les dates W3DTF

Il ne nous reste plus qu'à traiter les dates W3DTF. Commençons par compléter `adjustData`.

Listing 10-24 Notre fonction `adjustData`, terminée (fragment)

```
function adjustData(html) {
    var html = html.gsub(RE_TIMESTAMP, function(match) {
        match[2] = w3dtfToDate(match[2]).toLocaleString();
        return match[1] + match[2] + match[3];
    });
    DIV_OPENER = '<div class="content">';
    ...
} // adjustData
```

La fonction `w3dtfToDate` est un rien lourde... Nous utilisons une expression rationnelle pour découper le texte, un itérateur pour convertir les fragments numériques en

objets Number et une série d'invocations sur un objet Date tout frais. Il faut en plus gérer le décalage GMT.

Une date W3DTF a le format suivant :

```
yyyy-mm-dd[Thh:mm[:ss](Z|(+-)hh:mm)]
```

L'heure n'est pas obligatoire, et quand elle est là, elle ne précise pas forcément les secondes. Elle précise en revanche toujours un décalage GMT, soit avec Z (zéro décalage, donc GMT), soit sous format signé numérique. Dans notre expression rationnelle, nous considérons que l'heure et les secondes sont toujours là.

Pour prendre en compte le décalage, il faut stocker les données horaires comme s'il s'agissait de données GMT, puis appliquer un décalage opposé à celui indiqué. L'heure GMT stockée est ainsi correcte, et le formatage local, qui prend en compte *otre* fuseau horaire, affichera une heure dans *otre* référentiel.

Voici le code...

Listing 10-25 La fonction de conversion w3dtfToDate

```
RE_W3DTF = '^(\d{4})-(\d{2})-(\d{2})T(\d{2}):(\\d{2}):(\\d{2})'
    ↪ ([+-]\\d{2}:\\d{2}|Z)$';

function w3dtfToDate(text) {
    var comps = $A(text.match(RE_W3DTF)).map(function(s, index) {
        return 0 == index || 7 == index ? s : parseInt(s, 10); ①
    });
    var result = new Date();
    with (result) {
        setUTCFullYear(comps[1]);
        setUTCMonth(comps[2]);
        setUTCDate(comps[3]);
        setUTCHours(comps[4]);
        setUTCMinutes(comps[5]);
        setUTCSeconds(comps[6]);
    }
    var utcOffset = 0;
    if ('Z' != comps[7]) {
        var sign = '-' == comps[7].charAt(0) ? -1 : 1;
        var hours = parseInt(comps[7].substring(1, 3), 10);
        var minutes = parseInt(comps[7].substring(4), 10);
        utcOffset = sign * (hours * 60 + minutes);
    }
    result = new Date(result.getTime() - utcOffset * 60000);
    return result;
} // w3dtfToDate
```

L'exclusion des indices en ❶ repose sur le fait que l'indice 0 correspond à l'ensemble de l'expression (tout le texte W3DTF, qui ne nous est d'aucune utilité tel quel), et l'indice 7 est le décalage GMT, qui n'est jamais un simple nombre : nous voulons le conserver en tant que texte, pour analyse dans le bloc conditionnel (`if 'Z' != comps[7]...`).

Sauvons, rafraîchissons, et voici, comme sur la figure 10-8, de jolies dates en heure locale (qui est probablement la même que celle du Standblog : le fuseau horaire parisien).

Figure 10-8

Notre affichage de flux finalisé (texte grossi pour voir les dates/heures)



Pour aller plus loin...

Dans le cadre de ce chapitre, voici quelques références utiles.

Livres

RSS et Atom : Fils et syndication
Heinz Wittenbrink
Eyrolles, 8 juin 2006, 295 pages
ISBN 2-212-11934-8

Tout ce que vous avez toujours voulu savoir sur les formats de syndication, de leur histoire aux micro-détails techniques, avec tous les modules RSS 1.0 et RSS 2.0. Atom et APP sont également documentés. Tout est là. Vraiment tout.

Sites

- La RFC 4287 : Atom 1.0 et *Atom Publishing Protocol*
 - <http://tools.ietf.org/html/rfc4287>
- La « spécification » RSS 2.0
 - <http://blogs.law.harvard.edu/tech/rss>
- La spécification RSS 1.0 et ses modules
 - <http://web.resource.org/rss/1.0/>
- La spécification XSLT
 - <http://w3.org/TR/xslt/>

11

Mashups et API 100 % JavaScript

Au chapitre précédent, nous avons vu comment faire interagir nos pages avec des services externes, quitte à passer par un proxy sur notre serveur pour contourner les contraintes de sécurité imposées par les navigateurs. Cependant, de nombreux services sont aussi disponibles directement côté client, à l'aide de bibliothèques JavaScript fournies par des tiers. En combinant de tels services pour en fournir de nouveaux, on crée ce qu'on appelle désormais des *mashups*.

Dans ce chapitre, nous allons voir deux bibliothèques qui se prêtent particulièrement bien à une utilisation variée et intégrée dans tout type de page web : Google Maps pour utiliser des cartes (itinéraires, informations d'accès, etc.) et Google Chart, toute récente, pour réaliser rapidement des graphiques sur la base de données chiffrées.

Des cartes avec Google Maps

Commençons par les cartes. On ne présente évidemment plus Google Maps. Mais alors que la première génération de livres sur Ajax s'était, pour l'essentiel, attachée à recréer ce service, nous allons plutôt nous en servir, parce qu'il ne sert à rien de réinventer la roue (et si vous en avez tout de même envie, préférez les alternatives présentées à la fin de cette section).

Exigences techniques de l'API

Je précise deux points d'entrée de jeu :

- Google Maps fonctionne officiellement sur IE6+, Firefox 2+ et Safari 3.1+. Inutile donc d'espérer faire tourner ça sur des brontosaures comme IE5.5 ou Netscape 4, et il vous faudra vérifier le fonctionnement sur Opera 9 ou des navigateurs plus « exotiques » comme Konqueror ou iCab.
- Google Maps, comme la plupart des API Google, nécessite une clé API qui est étroitement associée au domaine (et souvent au numéro de port) sur lequel on va l'exploiter. Ce qui signifie au passage qu'il vous faudra mettre au point votre code sur un socle HTTP, et non en accès direct par fichier (le protocole `file://`).

Nous allons donc créer un répertoire de travail, par exemple nommé `gmaps`, et commencer par y placer une version minimale de notre `serveur.rb` simplement pour pouvoir accéder à nos fichiers en HTTP :

Listing 11-1 Le serveur HTTP minimaliste pour nos tests

```
#!/usr/bin/env ruby

require 'webrick'
include WEBrick

server = HTTPServer.new(:Port => 8042)
server.mount('/', HTTPServlet::FileHandler, '.')
trap('INT') { server.shutdown }
server.start
```

Décidément, vive Ruby... Nous déposons aussi un `prototype.js` classique, et allons pouvoir nous attaquer à Google Maps.

Il nous faut d'abord une clef API. Dans la mesure où nous allons tous, vous comme moi, utiliser `localhost:8042`, je vous livre directement la clef valable dans le code source du fichier HTML qui suit. Pour vos propres essais sur d'autres domaines, il vous faudra aller chercher une clef dédiée sur <http://code.google.com/apis/maps/signup.html>, ce qui nécessitera que vous ayez un compte Google (faute de quoi il faudra en créer un).

Voici le fichier HTML qui va nous servir pour nos tests, ainsi que la feuille de style minimalistique n'est là que pour définir les dimensions du conteneur pour la carte :

Listing 11-2 Notre index.html et le chargement de l'API

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr-FR" xml:lang=
"fr-FR">
<head>
    <meta http-equiv="Content-Type" content="text/html;
    charset=iso-8859-15" />
    <title>Jouons avec Google Maps</title>
    <link rel="stylesheet" type="text/css" href="gmaps.css" />
    <script type="text/javascript" src="prototype.js"></script>
    <script type="text/javascript"
        src="http://maps.google.com/maps?file=api&v=2&
        key=ABQIAAAuN4U8A21Hs7ruwZI9S1dTBS2R7FbmISiv4YEQqKkkC08_
        Li61RS6Bj1Zw5zGm1I4RpzYdqfhf2twYA"></script>
    <script type="text/javascript" src="gmaps.js"></script>
</head>
<body>

<h1>Jouons avec Google Maps</h1>

<div id="map"></div>

</body>
</html>
```

Observez la fourniture de la clef dans le paramètre `key` passé au chargement du script de l'API. On définit aussi un élément, généralement un `<div>`, qui va servir de conteneur à la carte. Ce conteneur doit absolument être dimensionné, car tout ce que l'API y mettra sera positionné en absolu, le conteneur étant par ailleurs en `overflow: hidden` et masquant donc ce qui dépasse. Faute de dimensions explicites, on ne verra donc... rien !

Listing 11-3 Dimensionner la carte dans gmaps.css

```
#map { width: 500px; height: 400px; }
```

Nous pouvons maintenant créer la carte.

Créer une carte

Tout d'abord, il est de bon ton de vérifier que le navigateur courant est bien compatible avec Google Maps ; on a pour ça la fonction `GBrowserIsCompatible()`.

On crée ensuite la carte au travers de l'objet `GMap2`, qu'on s'empresse d'initialiser avec un appel à `setCenter`. Cette fonction sert deux objectifs : positionner la carte et initialiser celle-ci. Elle doit être appelée avant toute autre manipulation !

Le constructeur `GMap2` prend juste comme argument l'élément conteneur. Attention, on n'est pas dans Prototype ou script.aculo.us : il faut passer l'élément lui-même, pas son identifiant.

```
var map = new GMap2($('map'));
```

Bon, en réalité on peut aussi passer un *hash* littéral d'options en deuxième argument, mais celles-ci sont assez peu intéressantes car elles peuvent facilement être réglées par des appels ultérieurs.

Tout de suite après, il faut l'initialiser avec un appel à `setCenter`, fonction qui prend deux arguments : la position du centre, sous forme d'un objet `GLatLng` (latitude et longitude), et le niveau de zoom (dont la valeur maximale varie d'un type de carte à l'autre ; par exemple, le type par défaut, un plan de ville, plafonne à 17, tandis que le type hybride plafonne à 19).

Calons-nous par exemple au-dessus de chez moi (eh oui !), et tant qu'à faire on va passer du simple plan à la carte hybride (plan + vue satellite). Vu que c'est à Paris, on peut pousser le niveau de zoom au maximum :

```
map.setCenter(new GLatLng(48.8828, 2.3222), 19);
map.setMapType(G_HYBRID_MAP);
```

Voyons ce que ça donne :



Figure 11-1 Notre carte basique (bienvenue chez moi)

Super ! Avant d'aller plus loin, permettons-nous tout de même de zoomer et dézoomer, au moins en faisant un double-clic (zoom), ou en utilisant la molette de la souris (zoom et « dézoom »), et pourquoi pas en zoom fluide (progressif au lieu d'être brusque)...

```
map.enableDoubleClickZoom();
map.enableScrollWheelZoom();
map.enableContinuousZoom();
```

Ajouter des contrôles et des marqueurs

Assez fréquemment, sur une carte Google, on désire ajouter des contrôles de navigation (zoom et déplacement), ainsi qu'une bascule entre différents types de cartes (généralement le mode plan et le mode hybride). On veut aussi parfois placer un ou plusieurs marqueurs, souvent associés à une infobulle HTML. Voyons comment réaliser rapidement tout ceci.

Des contrôles pour bouger et s'y retrouver

Une carte peut avoir de nombreux types de contrôles, qui sont des implémentations de `GControl`. On trouve notamment :

- `GSmallMapControl`, un de mes préférés, qui fournit des flèches de déplacement et deux boutons pour zoomer et dézoomer.
- `GLargeMapControl`, le même avec un curseur de zoom en plus.
- `GSmallZoomControl`, juste les boutons pour zoomer et dézoomer.
- `GScaleControl`, qui donne l'échelle.
- `GMapTypeControl`, qui fournit trois boutons, par défaut en haut à droite, pour le type de la carte : plan, satellite et hybride. On peut lui demander d'utiliser des libellés plus courts si nécessaire.
- `GMenuMapTypeControl` fournit le même service mais sous forme d'une liste déroulante, ce qui prend moins de place.
- `GHierarchicalMapTypeControl` groupe les types de cartes par fonctionnalité ; peu utile quand on se limite aux trois types de base, il trouve son intérêt lorsqu'on ajoute des types supplémentaires grâce à des services tiers (trafic, images issues de Flickr, etc.).
- `GOverviewMapControl` affiche dans un coin un petit bloc basculable (ouvert/fermé) qui situe la carte générale dans son contexte géographique.

Chaque contrôle prédéfini a une position par défaut, exprimée sous la forme d'un objet `GControlPosition`, lequel utilise un point d'ancrage (un des quatre angles de la carte) et un décalage (horizontal et vertical) par rapport à ce point. Les décalages sont exprimés sous forme d'objets `GSize`.

Afin d'illustrer tout cela, nous allons compléter notre carte avec les éléments suivants :

- des boutons pour choisir le type de carte, à leur position par défaut (en haut à droite) ;
- des boutons de déplacement et de zoom positionnés explicitement sous ceux de type de carte, plutôt qu'à leur position par défaut en haut à gauche ;
- une mini carte pour situer la vue dans un plan plus global ;
- l'échelle de la vue.

Voici le code :

```
map.addControl(new GMapTypeControl());
map.addControl(new GSmallMapControl(),
    new GControlPosition(G_ANCHOR_TOP_RIGHT, new GSize(7, 30)));
map.addControl(new GOverviewMapControl());
map.addControl(new GScaleControl(),
    new GControlPosition(G_ANCHOR_BOTTOM_LEFT, new GSize(5, 40)));
```

La figure 11-2 montre le résultat obtenu.



Figure 11-2 Notre carte avec quelques contrôles bien placés

Des marqueurs pour identifier des points spécifiques

La plupart du temps, on souhaite définir des positions particulières sur la carte, qu'on va représenter par des *marqueurs*. Un marqueur, c'est une latitude, une longitude, et potentiellement bon nombre d'options, de l'icône employée (laquelle revêt elle-même de nombreuses propriétés) à la possibilité de le glisser-déplacer, et j'en passe. C'est

aussi la possibilité d'ouvrir pour ce marqueur une infobulle plus ou moins complexe, contenant du HTML.

Commençons par un marqueur tout simple, qui va se placer à peu près sur mon immeuble :

```
var marker = new GMarker(new GLatLng(48.88281, 2.32212));
map.addOverlay(marker);
```

On crée un marqueur avec un objet `GMarker`, auquel on fournit au minimum une position. Un marqueur est considéré par la carte comme n'importe quel autre élément graphique additionnel qui ne serait pas un contrôle : comme un *overlay* (une sorte de calque). On utilise donc la fonction générique `addOverlay` de la carte pour l'ajouter à la vue.

La figure 11-3 montre notre marqueur, positionné pile au-dessus de mon appartement :



Figure 11-3 Un marqueur basique

Voyons rapidement comment y ajouter une infobulle personnalisée. Il suffit pour cela de réagir à certains événements en appelant la méthode `openInfoWindowHtml` (ou une variante plus avancée, par exemple avec des onglets...). Lorsqu'on souhaite simplement réagir au clic, on peut s'épargner des étapes en passant le HTML directement à `bindInfoWindowHtml`. Ces méthodes acceptent diverses options, `maxWidth` étant assez populaire (pour éviter que l'infobulle ne déborde trop en largeur).

Si l'on voulait juste réagir au clic pour afficher un fragment HTML, on pourrait par exemple faire :

```
marker.bindInfoWindowHtml('Et <strong>hop !</strong>');
```

Si l'on souhaite réagir à d'autres événements, il faut définir des gestionnaires pour ceux-ci. Voici un exemple qui affiche automatiquement l'infobulle lorsqu'on survole le marqueur :

```
GEvent.addListener(marker, 'mouseover', function() {
  marker.openInfoWindowHtml('Et <strong>hop !</strong>');
});
GEvent.addListener(marker, 'mouseout', function() {
  marker.closeInfoWindow();
});
```

On peut aussi rendre les marqueurs déplaçables par glissement, avec l'option `draggable`. On peut personnaliser en détail le comportement, par exemple le fait que le marqueur « rebondisse » quand on le lâche (option `bouncy`, à `true` par défaut). L'exemple suivant modifie notre marqueur pour qu'il affiche son infobulle uniquement quand on clique dessus ou quand on termine de le déplacer (c'est-à-dire au moment auquel il rebondira rapidement sur la carte).

```
var marker = new GMarker(new GLatLng(48.88281, 2.32212), {
  draggable: true
});
function showInfo() {
  marker.openInfoWindowHtml('Et <strong>hop !</strong>');
}
GEvent.addListener(marker, 'click', showInfo);
GEvent.addListener(marker, 'dragend', showInfo);
map.addOverlay(marker);
```

Ce n'est pas bien difficile, n'est-ce pas ? La figure 11-4 montre notre marqueur après un léger déplacement, son infobulle ouverte.



Figure 11-4 Le marqueur légèrement déplacé, avec son infobulle

Terminons cette rapide découverte de l'API Google Maps en voyant comment modifier l'icône employée pour notre marqueur. Une icône, représentée par l'objet `GIcon`, est en réalité un objet assez complexe, puisqu'on peut en définir indépendamment l'image de base, l'ombre, la taille, l'`« ancre »` (c'est-à-dire le point sur l'image qui est calé sur les coordonnées fournies), les versions transparentes et pour l'impression, l'image complémentaire pour le déplacement (par défaut une croix), etc.

Le plus simple consiste souvent à utiliser `G_DEFAULT_ICON` comme base et à changer, au minimum, l'image de base (si les tailles sont identiques, cela suffit amplement).

Le constructeur de `GIcon` prend généralement une icône de base (typiquement `G_DEFAULT_ICON`), et éventuellement l'image de base personnalisée. On définit ensuite si nécessaire les propriétés pour lesquelles on a besoin de modifier la valeur copiée depuis l'icône de base.

On trouve sur le Web d'innombrables jeux de marqueurs libres pour Google Maps. Google en fournit d'ailleurs plusieurs, déclinés par couleurs, dans son projet GMaps-Samples : <http://gmaps-samples.googlecode.com/svn/trunk/markers>. On va utiliser ici la version verte du marqueur par défaut, nommé `blank`. Voici le changement à apporter au début du code :

```
var icon = new GIcon(G_DEFAULT_ICON,
  'http://gmaps-samples.googlecode.com/svn/trunk/markers/green/
  blank.png');
var marker = new GMarker(new GLatLng(48.88281, 2.32212), {
  draggable: true, icon: icon
});
```

Une figure est un peu inutile, vu l'impression en noir et blanc...

À titre de résumé, voici le code complet de création de notre carte :

Listing 11-4 Le code complet de création de notre carte

```
function initMap() {
  if (GBrowserIsCompatible()) {
    // Initialization
    var map = new GMap2($('map'));
    map.setCenter(new GLatLng(48.8828, 2.3222), 19);
    map.setMapType(G_HYBRID_MAP);
    // Let the zooms begin!
    map.enableDoubleClickZoom();
    map.enableScrollWheelZoom();
    map.enableContinuousZoom();
    // A few controls
    map.addControl(new GMapTypeControl());
    map.addControl(new GSmallMapControl(),
      new GControlPosition(G_ANCHOR_TOP_RIGHT, new GSize(7, 30)));
    map.addControl(new GOverviewMapControl());
    map.addControl(new GScaleControl(),
      new GControlPosition(G_ANCHOR_BOTTOM_LEFT,
        new GSize(5, 40)));
    // And markers!
    var icon = new GIcon(G_DEFAULT_ICON,
      'http://gmaps-samples.googlecode.com/svn/trunk/markers/
      green/blank.png');
    var marker = new GMarker(new GLatLng(48.88281, 2.32212), {
      draggable: true, icon: icon
    });
    function showInfo() {
      marker.openInfoWindowHtml('Et <strong>hop !</strong>');
    }
  }
}
```

```
    GEvent.addListener(marker, 'click', showInfo);
    GEvent.addListener(marker, 'dragend', showInfo);
    map.addOverlay(marker);
}

document.observe('dom:loaded', initMap);
Event.observe(window, 'unload', GUnload);
```

Et tellement d'autres choses...

L'API GoogleMaps regorge de possibilités. Jetez donc un œil sur la référence : <http://code.google.com/apis/maps/documentation/reference.html>, c'est édifiant. On peut ajouter des contrôles (boutons, etc.), des *overlays* supplémentaires de tous types, des éléments graphiques (polygones, lignes, marqueurs...). Il est même possible de changer la projection utilisée pour le système de coordonnées (par défaut Mercator, celle dont a le plus l'habitude).

Saviez-vous d'ailleurs que Google Maps propose non seulement le globe terrestre, mais aussi des cartes satellitaires, infrarouges et topographiques de la Lune et de Mars, ainsi que de la voûte céleste ? Consultez donc la documentation du type `GMapType`, dont notre `G_HYBRID_MAP` n'est qu'une instance.

On peut aussi interagir avec de nombreux événements personnalisés, utiliser des services de géolocalisation pour situer des adresses, ou de trafic pour faire ressortir les conditions de circulation...

On peut enfin récupérer directement des données XML qui seront transformées avec du XSLT, et même incruster des mini-applications dans la carte (c'est la toute récente API Google Mapplets).

Et si l'on veut d'autres types de cartes ?

Eh bien, il n'y a pas que Google Maps dans la vie... On trouve aussi d'autres fournisseurs, comme Yahoo! Maps (<http://developer.yahoo.com/maps/>), et il existe même des API « universelles » qui fournissent une unification des concepts communs et permettent de modifier dynamiquement le service utilisé tout en gardant un code unique.

Au-delà de ces API, on peut tout bonnement ne retenir que certains morceaux de la pile (le système d'informations géographiques (GIS), le serveur de cartes, le cache de carreaux visuels et l'interface graphique supplémentaire) et remplir les blancs, comme le propose l'excellent article paru cette année sur l'incontournable A List Apart : *Take Control of Your Maps*, lisible sur www.alistapart.com/articles/takecontrolofyourmaps/. Une lecture utile, qui ouvre bien des horizons.

Des graphiques avec Google Chart

Google Chart est un service assez récent (proposé par Google depuis la fin 2007) qui permet d'obtenir instantanément des graphiques (au sens d'un graphique issu de tableau, pas d'un graphisme de jeu vidéo) de types très variés, potentiellement très complexes, sous forme d'images PNG optimisées.

On les intègre dans nos pages web, en laissant l'infrastructure Google s'occuper de la génération du graphique, de son envoi au navigateur (hop ! Une ressource en moins dans notre charge serveur !), de l'éventuelle mise en cache, etc.

Il est à noter que contrairement à de nombreuses API Google, celle-ci ne nécessite pas de clef API, et n'impose pas de limite particulière à l'utilisation, notamment en termes de volumétrie (au-delà d'un volume prévu de 250 000 requêtes quotidiennes, soit près de 3 requêtes/seconde en continu, prévenir tout de même Google en utilisant l'adresse de courriel indiquée dans la documentation).

Le principe

L'API Google Chart n'est pas à proprement parler du « 100 % JavaScript ». En effet, elle ne suppose... aucun JavaScript ! Car cette API repose entièrement sur l'utilisation de balises `` dont l'attribut `src=` contient un appel encodé au service Google Chart. Rien de plus !

Dans ce chapitre, nous n'allons donc même pas avoir besoin d'une page HTML pour nos tests : nous pouvons taper directement les URL de nos appels dans la barre d'adresse de notre navigateur, et *presto !* l'image apparaîtra. C'est bien pratique pour tester et garder une trace de nos tests : un simple dossier de marque-pages suffit...

Une URL de graphique Google Chart démarre nécessairement comme suit :

`http://chart.apis.google.com/chart?`

La suite est une série de paramètres URL-encodés, séparés par la traditionnelle esperluette (`&`). Je précise tout de même une évidence : comme toute URL, elle est « sur une seule ligne ». Au fil des paramètres de personnalisation, l'URL se rallonge évidemment, et sera donc souvent reproduite sur plusieurs lignes dans ce livre, mais vous devrez en pratique la taper d'une seule traite pour qu'elle fonctionne.

Trois paramètres sont incontournables :

- `chs (Chart Size)`, qui donne les dimensions en pixels de l'image (ces dimensions ne sont pas complètement libres, mais soumises à des valeurs maximales détaillées plus loin).
- `chd (Chart Data)`, qui fournit les données numériques sur lesquelles se base le graphique. L'API propose plusieurs manières d'encoder ces données suivant leur

nature et leur complexité, mais aussi en fonction des contraintes de longueur des URL obtenues.

- *cht* (*CHart Type*), qui indique le type de graphique à utiliser. Il existe de nombreux types de graphiques : histogrammes, lignes et *sparklines*, radar, nuage de points, diagrammes de Venn, camemberts, « Google-o-mètre » (un arc parcouru par une flèche, un peu comme un compteur de vitesse, ou tachymètre, sur une voiture) cartes géographiques et même « flash codes » (si, si !).

Tous les autres paramètres sont optionnels. L'API en fournit une quantité permettant de préciser les couleurs, gradients et motifs de remplissage et de trait, les axes avec leur titre, étiquette, marques de découpage et échelle, les titres et légendes de figure, les marqueurs de points de données, et j'en passe...

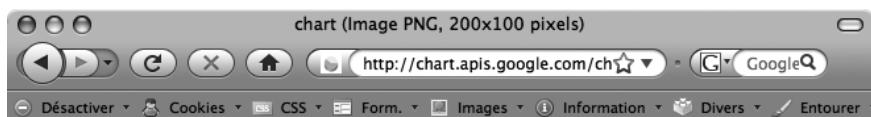
Créer un premier graphique

Commençons par un exemple tout simple, histoire d'avoir un peu de « gratification immédiate », comme disent les psychologues. Ouvrez une fenêtre ou un onglet dans votre navigateur, et saisissez dans la barre d'adresses l'URL suivante :

```
http://chart.apis.google.com/chart?chs=200x100&cht=p3&chd=t:60,40
```

On n'a ici mis que les trois paramètres obligatoires. Vous obtenez le graphique illustré en figure 11-5.

Figure 11-5
Un graphique rustique
(quoique 3D)



Décomposons l'appel :

- Le paramètre chs indique 200x100, soit un graphique occupant au total 200 pixels de large sur 100 de haut.
- Le paramètre cht a la valeur p3, qui signifie *Pie chart 3D*, c'est-à-dire un graphique « camembert » en 3 dimensions.
- Le paramètre chd dit t:60,40. Ce paramètre démarre toujours par le type d'encodage des données suivi d'un deux-points, comme nous le verrons tout à l'heure. Le type t indique l'encodage *textuel simple*, où les valeurs forcément positives (elles vont de 0 à 100) sont écrites directement, séparées par des virgules.

Le camembert s'est automatiquement centré au milieu de la zone qu'on lui a allouée (200×100), et les couleurs ont été choisies par l'API d'après sa palette par défaut. Juste histoire de pimenter un peu ce premier essai, ajoutons des étiquettes aux données avec le paramètre chl (*CHart Labels*), ce qui va nous obliger à élargir aussi le graphique pour les lire en entier :

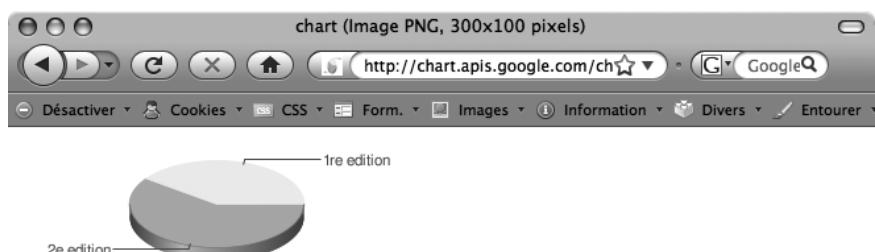
```
http://chart.apis.google.com/
chart?chs=300x100&cht=p3&chd=t:60,40&chl=2e%20edition|1ere%20edition
```

Évidemment, les caractères qui le nécessitent sont encodés, comme ici l'espace, remplacée par %20. Votre navigateur encode généralement pour vous, donc si vous tapez par exemple des espaces ou des caractères accentués, il les convertira tout seul dans la représentation encodée, ce qui vous facilite la tâche.

La figure 11-6 illustre le graphique ainsi étiqueté ; l'API positionne les étiquettes automatiquement en tentant de faire au mieux.

Figure 11-6

Notre graphique avec des étiquettes pour ses données



Les grands axes de personnalisation

Nous allons maintenant explorer les principales façons de personnaliser un graphique Google Chart. Toutefois, avant d'expérimenter avec les aspects visuels marquants, prenons le temps d'examiner les questions de taille et d'encodage des données, qui sont d'une importance capitale.

La taille

Un graphique Google Chart ne peut dépasser 300 000 pixels en général, et 440×220 dans le cas des cartographies. Hormis ce dernier cas donc, si vous optez pour une largeur de 500 pixels, la hauteur maximale sera de $300\,000 \div 500 = 600$ pixels.

C'est tout simple, mais il faut garder ces limitations à l'esprit. Autre point : les graphiques de type « camembert » exigent une certaine taille minimale pour leur lisibilité, calculée par l'API. Si la taille précisée est trop faible, le graphique sera tronqué plutôt que d'être trop réduit. La documentation nous informe à ce sujet qu'un camembert 2D (avec étiquettes) a généralement besoin d'un rapport largeur/hauteur de 2:1, tandis qu'un camembert 3D (idem) nécessite le plus souvent un rapport 2,5:1.

Le paramètre de taille, `chs`, est obligatoire et de la forme `largeurxhauteur`.

Les données

Les données sont la base du graphique, puisqu'elles sont les informations qu'on souhaite représenter graphiquement. Elles sont fournies par le paramètre obligatoire `chd`, qui commence par une lettre indiquant le type d'encodage retenu, suivie d'un deux-points, puis des données encodées.

Il existe quatre encodages possibles, parmi lesquels on choisira le sien suivant les besoins.

L'encodage textuel simple

C'est le plus naturel à lire, mais il utilise une échelle prédéfinie pour le graphique. On doit donc ajuster auparavant les valeurs en ramenant de l'intervalle [valeur minimale, valeur maximale] à l'intervalle [0, 100]. Un tel encodage démarre par `t:` suivi des valeurs numériques, éventuellement à virgules (enfin, des points en l'occurrence, puisque l'anglais utilise un point décimal et que la virgule sépare ici les valeurs).

S'il y a plusieurs séries de valeurs, ces séries sont séparées par un `pipe` (`|`). Les valeurs autorisées vont de 0 (zéro) à 100, -1 étant utilisé pour indiquer une valeur manquante ou inconnue.

Voici un encodage à une série :

```
t:1,2,3,6,7,14,21,42
```

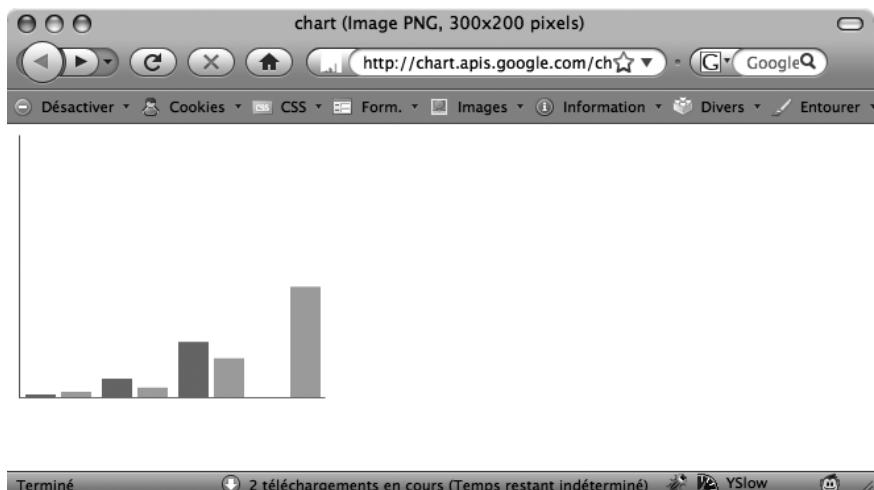
Et un autre à deux séries :

```
t:1,7,21|2,3.6,14.75,42
```

La figure 11-7 montre un graphique de type histogramme reposant simplement sur ce dernier encodage, d'après l'URL que voici (j'ai ajouté des codes couleur explicites pour les deux séries avec le paramètre chco, sans quoi il serait impossible de distinguer correctement les deux groupes).

```
http://chart.apis.google.com/chart?chs=300x200&cht=bvg&chd=t:1,7,21|2,3.6,14.75,42&chco=4d89f9,ff8800
```

Figure 11-7
Un histogramme à deux séries obtenu par encodage textuel simple



Il est à noter que sur ce type d'histogramme, les barres ont une largeur fixe, ce qui conditionne la taille de l'axe de données en fonction du nombre de valeurs dans les séries.

L'encodage textuel avec échelles : le plus universel mais le plus long

Dès qu'on veut sortir de l'échelle par défaut, par exemple pour ne pas avoir à convertir les valeurs, on utilise l'encodage textuel avec échelles. Il faut alors deux paramètres :

- chd pour les données brutes, évidemment. Il fonctionne exactement comme pour l'encodage textuel simple, mais les valeurs ne sont plus limitées à l'intervalle allant

de 0 à 100, donc -1 est une valeur comme une autre. Pour indiquer une valeur manquante, il faut en fournir une hors de l'échelle indiquée pour la série concernée.

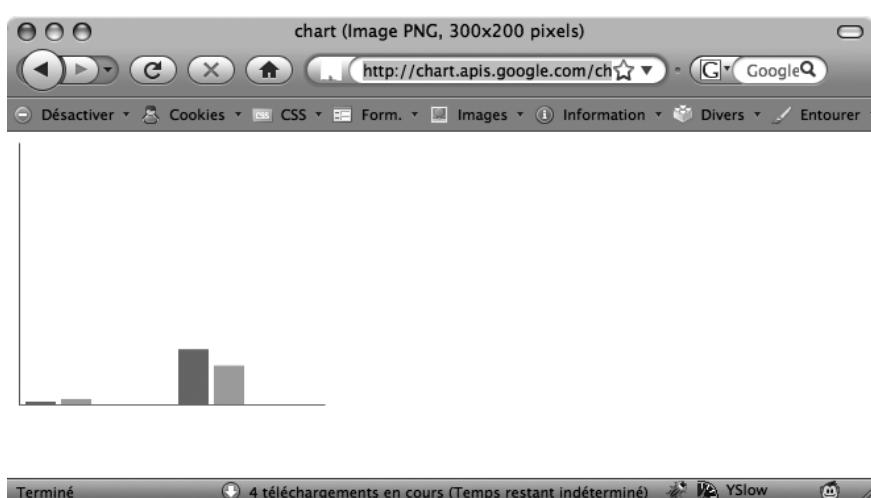
- `chds` (*CHart Data Scale*) pour les échelles des séries, exprimées sous forme de valeurs minimale et maximale. Chaque série de données dans `chd` reçoit donc deux valeurs limites, la minimale et la maximale, dans ce paramètre. Si l'on précise ici moins de paires que de séries dans `chd`, la dernière paire vaut pour les séries restantes. Par extension, préciser une seule paire de limites, donc une seule échelle, l'applique à toutes les séries de données.

Notez que ce type d'encodage n'est pas autorisé pour les cartographies.

Reprenons notre URL précédente et modifions-la un peu pour y placer quelques valeurs négatives, sans pour autant préciser d'échelle :

```
http://chart.apis.google.com/chart?chs=300x200&cht=bvg&chd=t:1,-7,21|2,-3.6,14.7,-42&chco=4d89f9,ff8800
```

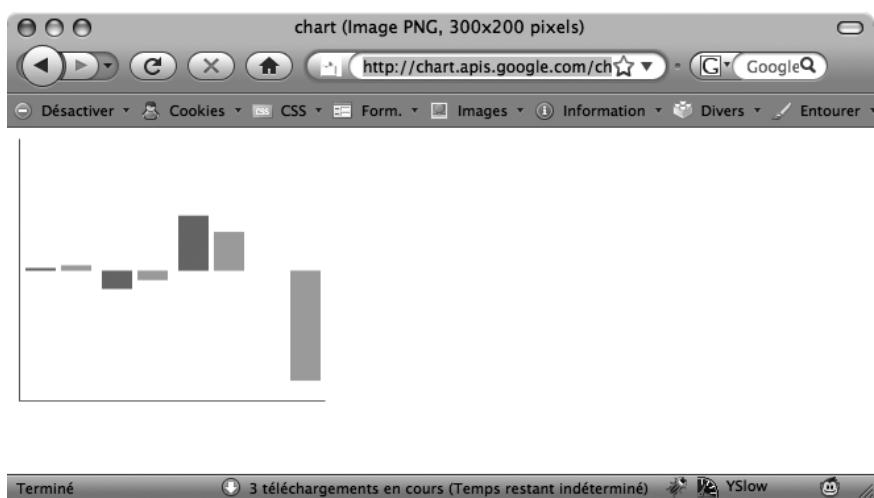
Figure 11–8
Un histogramme
avec des valeurs hors
de l'échelle par défaut



La figure 11–8 montre le résultat, qui ne correspond évidemment pas à nos attentes. En ajoutant le paramètre d'échelle, on obtient la figure 11–9 qui est correcte.

```
http://chart.apis.google.com/chart?chs=300x200&cht=bvg&chd=t:1,-7,21|2,-3.6,14.75,-42&chds=-50,50&chco=4d89f9,ff8800
```

Figure 11–9
Un histogramme
avec des échelles
personnalisées



L'axe horizontal n'a pas bougé pour se placer au centre (valeur zéro), parce que nous n'avons pas encore personnalisé les axes eux-mêmes, mais uniquement les échelles de valeur.

Ce type de description, avec des valeurs entières uniquement, permet déjà des graphiques de taille conséquente (500 pixels si l'on suppose l'habituel niveau de détail de 5 pixels par point de donnée), mais si l'on ajoute ne serait-ce qu'un niveau de décimale, on peut augmenter à l'envi le volume de données.

Notez en revanche que cet encodage, dans la mesure où il fournit jusqu'à 4 voire 5 caractères par valeur, plus les virgules et une description détaillée des échelles, est aussi celui qui produit les URL les plus longues.

C'est un problème car les URL sont soumises à diverses contraintes pour bien fonctionner, la plus restrictive étant celle de MSIE qui semble refuser des URL supérieures à 2 Ko (2 048 caractères), même si le serveur autorise plus (Apache permet en général 4 Ko, alors que des navigateurs comme Firefox, Safari ou Opera acceptent allégrement d'envoyer des URL immenses, de 64 Ko voire plus...).

Lorsqu'un graphique est susceptible de représenter un gros volume de données, un encodage plus compact est donc à favoriser. Nous allons en voir deux.

L'encodage simple : le plus court, mais pour de petites échelles

L'encodage simple consiste à décrire une valeur au moyen d'un seul caractère, ladite valeur étant donnée par la position du caractère dans la chaîne suivante :

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789

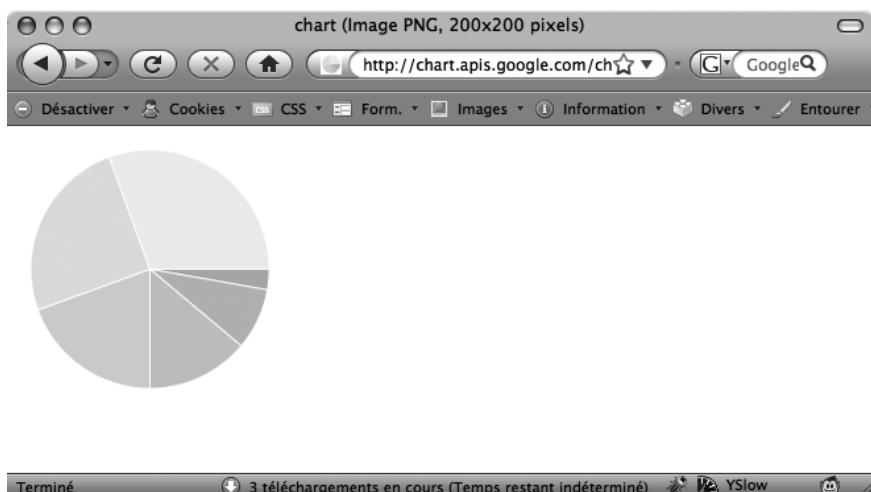
Ainsi, A vaut zéro tandis que 9 vaudra 61 (il y a 62 caractères). Puisqu'on sait qu'un caractère égale une valeur, plus besoin de virgules entre les valeurs : elles sont alors utilisées pour séparer les séries, à la place du *pipe*. Une valeur inconnue ou manquante est représentée par un *underscore* (_).

L'encodage simple démarre par le code s::

La figure 11-10 propose un camembert 2D avec des valeurs de 5, 15, 25, 35, 45 et 55, en conservant une échelle directe (c'est-à-dire sans l'ajustement d'échelle qui aurait lieu si les données d'origine utilisaient une autre échelle que [0,61]) :

`http://chart.apis.google.com/chart?chs=200x200&cht=p&chd=s:FPZjt3`

Figure 11-10
Un camembert obtenu
par un encodage
simple très compact



C'est extrêmement compact, d'autant qu'on n'a évidemment pas d'échelle puisqu'on est forcément entre 0 et 61. En revanche, il faut une conversion voire des arrondis, certes simples à programmer. Par ailleurs, cette échelle ne convient pas forcément à tous les besoins : si l'on souhaite représenter des données plus fines, il faut passer à l'encodage étendu.

L'encodage étendu : idéal pour les graphiques « lourds »

L'encodage étendu reprend le principe de l'encodage simple, avec deux ajustements.

- On ajoute deux caractères en fin de série pour aboutir à 64 positions (de 0 à 63) : le tiret (-) et le point (.) .
- Chaque valeur est codée par *deux* caractères ; on a donc des nombres à deux chiffres en base 64, soit $64^2 = 4\,096$ valeurs (2^{12}). Une valeur manquante est représentée par un *double underscore* (_). C'est là une résolution suffisante pour tous les types

de graphiques, même « lourds », c'est-à-dire riches en données d'amplitude variée. En revanche, l'URL obtenue voit doubler sa partie de données (ce qui reste néanmoins bien plus efficace en moyenne que l'encodage textuel).

L'encodage étendu des données démarre par `e:`.

À titre d'exemple, voici deux séries de valeurs : 143, 17, 2 151 et 418 d'une part, et 253, 997, 612 et 43 d'autre part (toujours sans ajustement d'échelle, pour faciliter la compréhension) :

```
| chd=e:CPARhnGi,D9P1JkAr
```

Ça reste quand même très compact...

Le type de graphique

Nous n'allons pas tout illustrer : la page globale de documentation de l'API le fait bien mieux que nous, et ce n'est pas notre propos. Mais un rapide tour d'horizon permet de situer les possibilités.

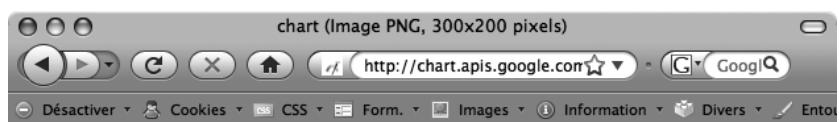
Il existe de nombreux types et sous-types de graphiques. Certains exigent des paramètres supplémentaires, suivant leur nature. Les principaux types sont :

- Les lignes de données. Soit on fournit juste des ordonnées et alors les abscisses sont réparties de façon homogène (type `1c`), soit on fournit pour chaque point de donnée *deux* valeurs : son abscisse et son ordonnée (type `1xy`).
- Les *sparklines* (par exemple un cardiogramme), du type `1s`, se comportent exactement comme des graphiques de type ligne, mais sans les axes par défaut.
- Les histogrammes ont de nombreuses variantes, suivant qu'on empile les séries sur une même ligne/colonne (types `bhs` et `bvs`), ou qu'on groupe les barres par position dans les séries (types `bhg` et `bvg`) comme nous l'avons fait dans ce chapitre. On contrôle également l'orientation horizontale (`bhs`, `bhg`) ou verticale (`bvs`, `bvg`). Un paramètre optionnel `chbh` permet de préciser la largeur des barres, l'espace entre barres d'un même groupe et l'espace entre les groupes de barres, plutôt que de laisser l'API utiliser ses valeurs par défaut qui pourraient aboutir à une troncation du graphique à la taille fournie par `chs`.
- Les camemberts existent en 2D (`p` pour *pie*) et 3D (`p3`). Évidemment, un camembert n'a qu'une série de données.
- Les diagrammes de Venn (type `v`) représentent des surfaces circulaires qui s'intersectent plus ou moins. On s'en sert souvent pour illustrer l'interpénétration de concepts, de profils clients, de marchés... On n'autorise ici que trois disques ! Le jeu de données a une structure fixe et très particulière, qui précise les tailles et surfaces d'intersection de ces disques.

- Les nuages de points (type s pour *scatterplot*) sont surtout utilisés en statistique pour examiner les « densités » de zones de valeur, par exemple en dégageant des tendances sur le temps passé devant la télé en fonction de l'âge...
- Le type radar (code r) définit des parcours autour d'une origine, suivant un azimuth et une distance. Si vous n'en avez jamais utilisé, vous n'en aurez probablement pas besoin.
- Le type cartographique (type t, je ne sais pourquoi...) est extrêmement utile et fonctionne d'une façon propre, que nous examinerons plus loin dans une section dédiée.
- Amusant, le « Google-o-mètre » représente une jauge en arc de cercle dont on peut contrôler la coloration et sur laquelle on positionne un indicateur, par exemple une flèche issue du centre géométrique de l'arc. La figure 11-11 en montre un exemple, obtenu à partir de l'URL suivante :

`http://chart.apis.google.com/
chart?chs=300x200&cht=gom&chd=t:70&chl=Qualit%C3%A9%20web`

Figure 11-11
Un exemple de jauge
type « Google-o-mètre »



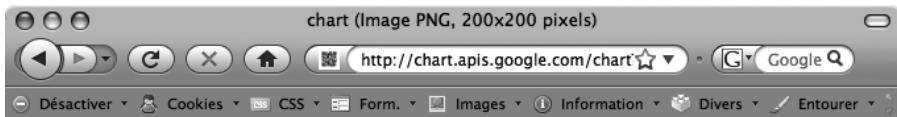
- Pour finir, le tout nouveau type qr (pour *QR code*, QR voulant dire à l'origine *Quick Response*) permet de définir des codes à barres bidimensionnels, les fameux « flash codes » qu'on trouve de plus en plus dans des publicités et affichettes, exploitables par exemple par les capacités photographiques des téléphones mobiles. Un tel graphique encode un texte (une URL, une adresse de courriel, des coordonnées géographiques...), fourni cette fois par ch1 au lieu de chd, allant jusqu'à 4 296 caractères (en supposant des caractères ASCII classiques ; en UTF-8, compter près de 3 000 caractères, ce qui est déjà confortable). En fait, le nombre de lignes et colonnes visuelles (pas de pixels, hein...) du graphique dépend directement de la taille du

texte à encoder. La figure 11-12 encode par exemple « Bien développer pour le Web 2.0, 2^e édition », et est obtenue avec l'URL que voici :

`http://chart.apis.google.com/chart?chs=200x200&cht=qr&chl=Bien%20d%C3%A9velopper%20pour%20le%20Web%202.0,%202e%20%C3%A9dition`

Figure 11-12

Le flash code (QR Code) du titre de cet ouvrage !



Les couleurs

Les couleurs dans Google Chart sont obligatoirement exprimées sous forme de codes hexadécimaux *à six chiffres*. La forme raccourcie autorisée par CSS, à trois chiffres (que je préfère) n'est pas disponible ici.

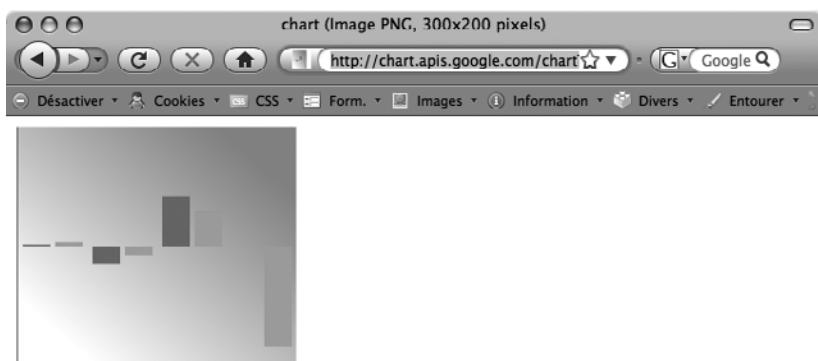
Google Chart permet de définir les couleurs pour cinq types d'éléments :

- Les couleurs du graphique lui-même, c'est-à-dire des données. Par exemple, celles des lignes, des barres d'histogramme, des disques d'un diagramme de Venn, des « parts » de camembert ou de l'arc du Google-o-mètre. C'est le paramètre `chco`, qui prend une série de codes couleur séparés par des virgules. Suivant le type de graphe, la correspondance de ces codes couleur aux données varie (la documentation en ligne en illustre brillamment les possibilités).
- Les couleurs de remplissage, utilisées pour « peindre » les surfaces entre deux lignes (ou entre les bords du graphe et une ligne). C'est le paramètre `chm`, qui peut avoir des valeurs assez complexes.
- Les couleurs ou gradients de fond (paramètre `chf`, pour *CHart Fill*). On a en fait deux fonds : l'image totale (bords, légendes, titres et étiquettes d'axes inclus) et le graphique lui-même (à l'intérieur des éventuels axes), et l'on peut définir des valeurs séparément pour chacun des deux, avec des options de transparence (mais on peut buter à ce moment-là sur la prise en charge par MSIE 6...).

- Au lieu d'une couleur unie, on peut par ailleurs demander un gradient linéaire, ou dégradé, d'un angle quelconque (de 0° à 90°), défini de façon conventionnelle (série de couleurs avec leur position de référence dans le dégradé, 0 étant à gauche, 1 à droite — contrairement à ce qu'annonce la documentation). La figure 11-13 illustre cette possibilité avec un dégradé (sous-type `1g`) blanc-bleu partant de l'origine (angle de 45°) pour la zone du graphique lui-même (type principal `c`) :

```
http://chart.apis.google.com/chart?chs=300x200&cht=bvg&chd=t:1,-7,21|2,-3.6,14.75,-42&chds=-50,50&chco=4d89f9,ff8800&chf=c,1g,45,ffffff,0,76a4fb,1
```

Figure 11-13
Notre histogramme à échelle personnalisée... avec un dégradé de fond



- Alternativement, on peut choisir des bandes de couleur, qu'on peut concevoir comme des sortes de hachures épaisses. Le sous-type est `1s` (*linear stripes*) au lieu de `1g` (*linear gradient*), et l'on précise un angle puis une série de barres avec leur couleur et leur épaisseur. Cette série est répétée jusqu'à remplir la surface demandée (le graphique lui-même ou toute l'image, comme d'habitude). On s'en sert généralement pour mettre en avant une ou plusieurs plage(s) de valeurs sur un des axes, mettant ainsi en exergue la progression des données sur ces plages.

Mais aussi...

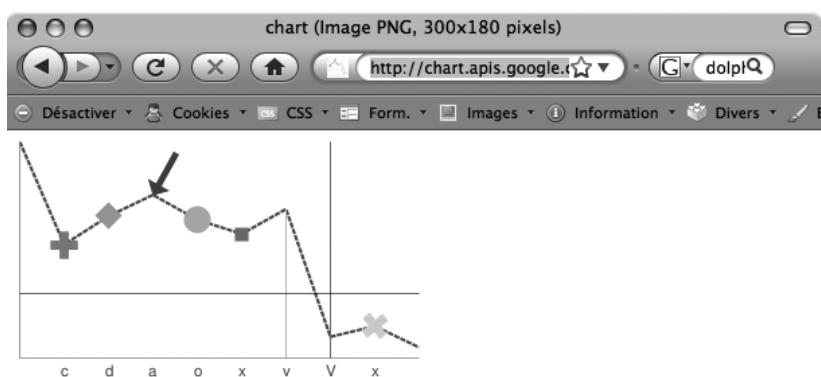
La place nous manque pour illustrer complètement les possibilités de Google Chart, mais la documentation en ligne, encore une fois regroupée sur une seule page, est excellente et très bien illustrée. Je vous fais juste un petit résumé des personnalisations restantes :

- Étiquettes : titre et légende du graphique, étiquettes des données ou séries de données...

- Axes : nombre, position, échelle, étiquettes et graduation, couleurs et polices de caractères...
- Styles de trait : épaisseur, motif de trait (plein ou alterné en précisant les longueurs de transparence et de solidité, ce qui est très flexible).
- Styles de grille : intervalles vertical et horizontal de la grille, motif de trait des lignes.
- Marqueurs : mise en avant de points de données ou d'intervalles de données sur les séries figurant dans chd. Chaque point peut se voir doté d'un marqueur graphique (flèche, croix, losange, cercle, carré, X), textuel ou comme ligne ramenant à un axe, avec une couleur et une « priorité », utilisée en cas de superposition de marqueurs. Ça peut donner des trucs hallucinants, comme le montre la figure 11-14.

Figure 11-14

Des marqueurs de points de données en pagaille !



Bon, évidemment, l'URL fait un peu froid dans le dos...

```
http://chart.apis.google.com/chart?cht=lc&chd=s:9gounjqGJD&chco=
008000&chl=s=2.0,4.0,1.0&chs=300x180&chxt=x&chxl=0:||c|d|a|o|x|v|V|x|
&chm=a,990066,0,3.0,9.0|c,FF0000,0,1.0,20.0|d,80C65A,0,2.0,20.0|o,
FF9900,0,4.0,20.0|s,3399CC,0,5.0,10.0|v,BBCCED,0,6.0,1.0|V,3399CC,0,
7.0,1.0|x,FFCC33,0,8.0,20.0|h,000000,0,0.30,0.5
```

Carte ou graphique ? Les cartographies

Lorsqu'on doit représenter des données relatives à des pays, le mode cartographique est tout simplement parfait ! Il permet de choisir une carte parmi plusieurs régions du monde (ou de choisir le monde entier), et de définir un dégradé de couleurs au sein duquel on va piocher pour colorer des pays en fonction de la valeur qu'y prend la grandeur à représenter.

Le type d'encodage utilisé pour les données (l'habituel paramètre chd) détermine l'échelle de valeurs (textuel simple : 0 à 100 ; simple : 0 à 61 ; étendu : 0 à 4095 ; le type textuel avec échelles n'étant pas autorisé).

Une cartographie utilise les paramètres suivants :

- cht=t (type cartographique).
- chtm pour la région représentée. Valeurs possibles : world, europe, usa, africa, asia, middle_east, south_america.
- chs pour la taille (440 × 220 au maximum).
- chco pour les couleurs associées aux pays en fonction de leur valeur. On commence par une couleur par défaut (pays non listés comme étant « à colorier »), puis les deux extrêmes d'un dégradé linéaire de couleurs (couleurs pour la valeur minimale et la valeur maximale de l'échelle des données).
- ch1d pour la liste des codes ISO 3166-1 alpha-2 de pays à colorier (par exemple FR, GB, DE, ES...), concaténés sans virgules vu qu'ils ont une taille fixe de 2 caractères.
- chd pour les données (encodage textuel simple, encodage simple ou encodage étendu), à raison d'une valeur par pays à colorier.

Prenons un exemple avec l'Europe, qui aurait des pays en gris très pâle par défaut, un gradient du vert très clair au vert profond, et des valeurs de population ramenées à l'échelle, en encodage simple, pour la France, la Suède et l'Allemagne.

- L'encodage simple prévoit 62 valeurs, de 0 à 61, soit de A à 9. On va caler sur 61 la plus forte population pour les trois pays retenus, soit celle de l'Allemagne.
- Wikipedia m'indique qu'au dernier recensement...
 - la France avait 64 473 140 habitants, soit une valeur à l'échelle de $64\ 473\ 140 \div 82\ 314\ 906 \times 61 = 47,7$ arrondi à 48, encodé en w ;
 - la Suède avait 9 208 034 habitants, soit une valeur à l'échelle de 6,8 ramenée à 7, encodée en G ;
 - l'Allemagne avait 82 314 906 habitants, soit 61 à l'échelle, encodée en 9.
- Les codes ISO-3166-1 alpha-2 sont respectivement FR, SE et DE.

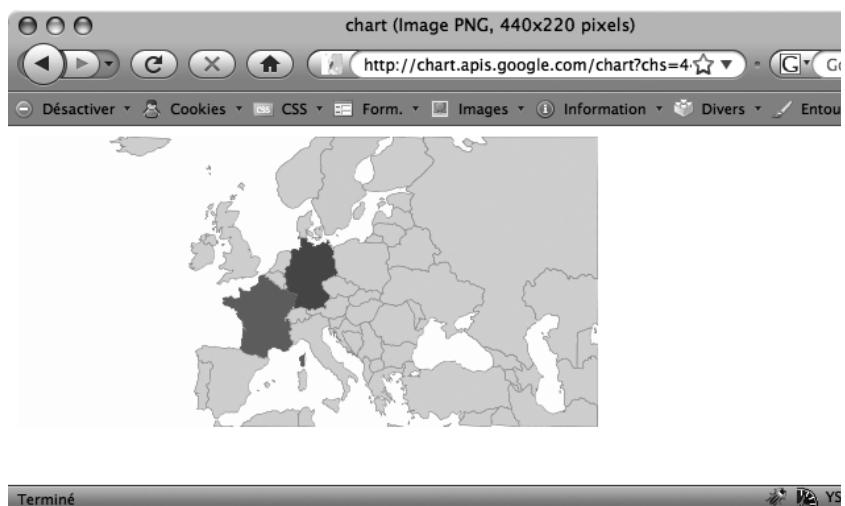
Voici donc l'URL :

```
http://chart.apis.google.com/chart?chs=440x220&cht=t&chtm=europe&chco=dddddd,bbffbb,008000&ch1d=FRSEDE&chd=s:wG9
```

La figure 11-15 illustre le résultat.

Figure 11-15

L'Europe avec trois pays coloriés d'après leur population



Sympathique, n'est-ce pas ?

Alternatives suivant les besoins

Si vous avez besoin de réaliser des graphiques pour vos pages web tout en restant côté client, Google Chart n'est pas la seule option.

J'aime beaucoup Flotr, par exemple, un portage JavaScript basé sur Prototype de la bibliothèque Flot, réalisé par les super p'tits gars de Solutoire. Elle permet notamment, contrairement à Google Chart :

- d'appliquer des fonctions mathématiques ;
- de gérer une certaine interactivité avec le graphique, en suivant la souris pour mettre en avant des points de données, en gérant le zoom par sélection de zone à la souris, etc. On peut aussi passer les données au format JSON, ce qui évite d'avoir à faire trop d'encodage...

Voici par exemple une petite création de graphique basé sur fonctions, avec une légende :

```
function myLabelFunc(label){  
    return 'y = ' + label;  
}  
  
var f = Flotr.draw(  
    $('#container'), [  
        {data:d1, label:'x + sin(x + ?)'}, // Un vrai Pi !
```

```

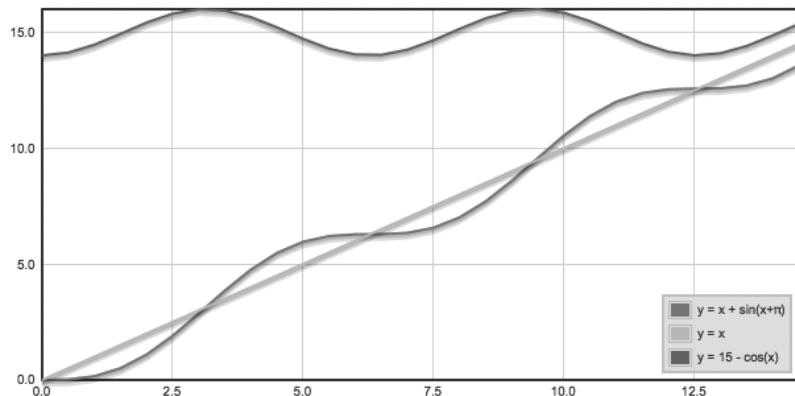
        {data:d2, label:'x'},
        {data:d3, label:'15 - cos(x)'}
    ], {
        legend:{
            position: 'se', // Légende en bas à droite
            labelFormatter: myLabelFunc, // Formatage étiquettes
            backgroundColor: '#d2e8ff' // Fond bleu clair
        }
    }
);

```

La figure 11-16 montre le résultat obtenu.

Figure 11-16

Un graphique basé sur fonctions mathématiques avec Flot



D'ailleurs, chez Solutoire, ils aiment tellement le domaine des graphiques qu'ils ont même créé un enrobage JavaScript autour de Google Chart ! Si vous pensez devoir recourir dynamiquement côté client à Google Chart plutôt que de préparer l'URL côté serveur, cet enrobage peut vous être utile. Toutefois, il n'a pas été mis à jour depuis décembre 2007, il lui manque donc quelques petites nouveautés, mais c'est facile à ajouter...

Évitez en revanche d'en passer par là si vous pouvez déterminer le graphique côté serveur : en évitant JavaScript, vous gagnez en accessibilité si ce dernier est désactivé sur le navigateur...

Et puis, il existe divers enrobages *côté serveur* de Google Chart, pour générer l'URL pour vous. Par exemple, en Ruby on Rails, on a notamment Google Chart On Rails, qui fait que dans nos vues on retrouve du code du genre :

```
GoogleChart.pie(10,20,40,30).to_url
```

Moi je dis : « ça va... »

Pour aller plus loin

Sites

- Tout savoir sur Google Maps : obtenir une clef API, lire la documentation, consulter les innombrables exemples et didacticiels :
<http://code.google.com/apis/maps/>.
- L'article révélateur sur A List Apart pour découvrir les systèmes alternatifs existants lorsqu'on veut créer des cartes personnalisées et complexes :
<http://www.alistapart.com/articles/takecontrolofyourmaps/>.
- Tout savoir sur Google Chart, qui a aujourd'hui l'avantage de disposer d'une documentation presque en une seule page, bien que très complète :
<http://code.google.com/apis/chart/>.
- Flotr, la bibliothèque sympathique de Solutoire : <http://solutoire.com/flotr/>.
- GChart, l'enrobage JavaScript de Google Chart, toujours chez Solutoire :
<http://solutoire.com/gchart/>.
- Google Chart on Rails :
<http://code.google.com/p/google-charts-on-rails/>
- Flotr, l'API 100 % JavaScript, basée sur Prototype, pour faire des graphiques interactifs avancés :
<http://solutoire.com/flotr/>.

ANNEXES

Annexe A

Bien baliser votre contenu : XHTML sémantique

Annexe B

Aspect irréprochable et flexible : CSS 2.1

Annexe C

Le « plus » de l'expert : savoir lire une spécification

Annexe D

Développer avec son navigateur web

Annexe E

Tour d'horizon des autres frameworks JavaScript

A

Bien baliser votre contenu : XHTML sémantique

Au commencement était le contenu. La toute première étape de la création de votre page web doit être ce contenu. Il n'y a pas plus important. C'est lui qui fait la différence entre votre page et les autres. C'est ce contenu que voient avant tout les moteurs de recherche et les périphériques alternatifs. C'est lui qui définit la valeur, le sens et l'identité de votre page. Vous avez le devoir, je dis bien le devoir, de lui consacrer tout le soin possible.

Ce n'est qu'une fois votre contenu défini et mis en place que vous pouvez vous poser la question de son aspect et de son comportement. L'aspect sera le rôle des CSS, et nous verrons à l'annexe B que leur usage ne nécessite qu'extrêmement peu d'intrusions dans votre contenu pour satisfaire vos envies les plus variées. Quant au comportement, ce sera le rôle de JavaScript, et si vous devez retenir une chose des chapitres 2 et 3, c'est que son utilisation n'entache pas non plus votre contenu.

Cette annexe commence par présenter rapidement les nombreux bénéfices qu'apporte la présence de contenus « propres » dans vos pages web, c'est-à-dire, pour l'essentiel, des contenus valides et surtout sémantiques. Ensuite, vous trouverez un court passage en revue des contraintes syntaxiques, des éléments disponibles et de leur utilisation pertinente, pour finir par quelques exemples concrets.

Les avantages insoupçonnés

Prendre les bonnes habitudes pour créer un contenu de qualité ne rend pas le travail plus difficile (bien au contraire), mais apporte de nombreux bénéfices, parfois inattendus.

Pour le site et ses propriétaires

En tant que commanditaire d'un site web, qu'avez-vous à gagner à ce qu'il soit réalisé en XHTML 1 Strict, et à ce qu'il utilise un balisage rigoureusement sémantique ?

Beaucoup de choses :

- 1 Il sera infiniment plus compréhensible par les moteurs d'indexation et de recherche. Par conséquent, votre visibilité sur Google et autres, mais surtout la pertinence de cette visibilité, seront grandement améliorées.
- 2 Le balisage sémantique entraîne automatiquement une accessibilité accrue, ce qui signifie que votre site est consultable par un plus large public. Il s'agit d'abord des internautes atteints d'un handicap (visuel, moteur, cognitif), ce qui vous met en conformité avec les lois présentes ou à venir quant à l'accès aux sites web pour ces personnes.
Il s'agit aussi des visiteurs utilisant autre chose qu'un ordinateur avec un grand écran, un clavier et une souris : ceux navigant sur votre site à l'aide d'un Palm, d'un PocketPC, d'un TabletPC, d'un téléphone 3G, d'un Blackberry, etc., ont par nécessité une préférence marquée pour les sites accessibles. Ils constituent qui plus est une cible marketing intéressante (adeptes de haute technologie et disposant de moyens financiers suffisants pour se les acheter).
- 3 Un tel site « pèse » beaucoup moins lourd en termes d'octets. D'innombrables refontes complètes de sites appliquant ces préceptes ont abouti à un contenu tout aussi riche mais beaucoup moins lourd. Conséquence directe : la consommation de bande passante (et donc son coût) diminue sensiblement, alors même que la fréquentation augmente (et donc les revenus aussi).
- 4 La fréquentation du site augmente, car il n'est plus nécessaire de maintenir différentes versions du contenu : l'utilisation des CSS permet de s'adapter automatiquement d'un périphérique de consultation à l'autre. Oublié, le temps où les versions alternatives (pour impression ou pour non-voyants) étaient souvent en retard sur la version principale ! Non seulement votre contenu est à jour pour tous, mais il devient accessible à une base plus large d'utilisateurs. Par ailleurs, cette centralisation du contenu signifie que sa mise à jour est plus simple et plus rapide : vous gagnez en parts, mais aussi en réactivité !
Ce cercle vertueux ne peut que générer des revenus supplémentaires, qui ont tôt fait d'amortir le coût d'une éventuelle refonte complète du site.

Vous retrouverez ces arguments sous un développement différent dans la section « À quoi servent les standards du Web ? » de l'avant-propos. Il contient notamment un lien vers l'exemple célèbre du site de la chaîne de sports américaine ESPN, qui constitue un cas d'école magistral, avec des chiffres impressionnantes à l'appui.

Pour le développeur web

Les développeurs chargés d'implémenter un site respectant ces principes de qualité ont aussi la vie plus facile.

- 1 Un document XHTML ne présente aucune ambiguïté, par comparaison à un document HTML classique. En y plaçant la bonne déclaration DOCTYPE, on peut tirer parti des validateurs en ligne pour vérifier la conformité du document à sa grammaire imposée, sans risque d'erreur. Des validateurs identiques existent pour les CSS, par exemple.
Si les propriétaires du site imposent des règles éditoriales qui affectent la grammaire du document, il est possible de réaliser une DTD ou un schéma transcrivant ces contraintes pour bénéficier des mêmes tests automatisés de conformité.
- 2 Un des avantages de l'univers XML est qu'il permet, grâce au mécanisme des espaces de noms, de mélanger plusieurs vocabulaires dans un même document. Ainsi, en XHTML, on peut insérer des fragments de langages à balises supplémentaires, comme SVG pour les graphiques vectoriels, MathML pour les formules mathématiques ou encore SMIL pour le multimédia.
- 3 Un document sémantique a par définition beaucoup moins de balises qu'un document des années 1990, et beaucoup plus d'attributs id correctement définis : cela le rend plus simple à habiller avec les CSS, et plus simple à scripter en terme de DOM.

Règles syntaxiques et sémantiques

XHTML 1 Strict est la variante stricte de XHTML. XHTML lui-même peut se résumer à HTML 4.01 auquel on applique les règles syntaxiques de XML. Ces distinctions sont signalées dans la section 4 de la recommandation pour XHTML 1.0, et elles sont très simples.

- **Les documents doivent être correctement formés.** Cela signifie essentiellement que les balises doivent se fermer dans l'ordre inverse de leur ouverture : `<i>...</i>...` est correct, mais `<i>......</i>` ne l'est pas. Les balises ne sont pas comme des commutateurs, mais constituent véritablement des conteneurs : on doit pouvoir déterminer quel élément est dans quel autre.

- **Par extension, tout élément doit être fermé.** En HTML, de nombreux éléments pourtant non vides n'étaient pas fermés par négligence : principalement `li` et `option`, mais aussi trop souvent `p`. On ferme un élément avec sa balise classique précédée d'un `/` : `<p>` devient `</p>`.
- **Les noms d'éléments et d'attributs doivent être en minuscules.** En XML, on est sensible à la casse, et les grammaires formelles pour XHTML utilisent les minuscules (c'est par ailleurs plus agréable visuellement).
- **Les éléments vides doivent aussi être fermés.** Il s'agit des éléments `area`, `base`, `br`, `col`, `frame`, `hr`, `img`, `input`, `link`, `meta` et `param`. Pour fermer un élément vide en XML, on ajoute simplement un `/` avant son chevron fermant. Toutefois, certains vieux navigateurs sont perturbés par cette notation ; aussi prend-on généralement la précaution d'ajouter une espace devant le `/`, comme ceci :
``.
- **Les valeurs d'attributs sont entre guillemets.** Techniquement, XML autorise tant les apostrophes (`'`) que les guillemets (`"`). Je recommande vivement, à titre personnel, la deuxième possibilité.
- **Les attributs bascules ont tout de même une valeur.** En HTML, certains attributs sont obligatoires, mais sans qu'il soit nécessaire de préciser une valeur. C'est le cas notamment des attributs `checked`, `compact`, `disabled`, `readonly` et `selected`. En XML, un attribut a forcément une valeur. XHTML définit pour ces attributs une seule valeur autorisée, qui est leur propre nom. On écrira donc par exemple le code suivant : `<option value="carrier" selected="selected">Express</option>`.
- **L'attribut `name` n'est plus autorisé pour les éléments suivants :** `a`, `applet`, `form`, `frame`, `iframe`, `img` et `map`. En XML, le moyen standard d'identification d'un élément, quel qu'il soit, est l'attribut `id`. L'attribut `name` reste valide sur les autres éléments, notamment les champs de formulaires.
- Les valeurs par défaut des attributs sont fournies en minuscules uniquement (par exemple, `get` pour l'attribut `method` de `form`), toujours en raison de la sensibilité de XML à la casse.
- Un dernier détail : si vous utilisez des entités numériques avec un code hexadécimal, en XHTML le `x` initial est forcément en minuscules : `…` et non `…`.

La DTD et le prologue

On prendra également soin de toujours référencer la DTD en début de document, avant l'élément ouvrant `html`. Un contenu de qualité utilise soit la variante `Strict`, soit la variante `Frameset` si vous avez des cadres (frames). Toutefois, les cadres sont généralement considérés comme problématiques en termes d'accessibilité et de scripting. Ne les utilisez que si vous ne pouvez absolument pas faire autrement. Voici la déclaration DOCTYPE qui devrait figurer en haut de tous vos documents :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Cette déclaration n'a rien de superflu : sur la plupart des navigateurs, référencer une DTD stricte déclenche ce qu'on appelle le *doctype switching*, en forçant le navigateur à se placer en mode « respect des standards », ce qui change beaucoup de choses, notamment pour MSIE. D'ailleurs, toutes les améliorations CSS de MSIE 7 ne fonctionneront que dans ce mode, exigeant donc une déclaration de DTD stricte à l'ouverture du document.

Petit point de détail : un document XML (donc XHTML) utilisant un encodage autre que UTF-8 est censé le signaler d'entrée de jeu dans son prologue. Il s'agit d'une syntaxe spéciale présente dès le premier octet du document, et qui ressemble à ceci :

```
<?xml version="1.0" encoding="iso-8859-15"?>
```

Toutefois, MSIE 6 ne fonctionne pas correctement lorsqu'il reçoit un tel document servi comme du (X)HTML (ce point est corrigé dans MSIE 7). Aussi, on s'abstient pour l'instant de fournir un prologue. Heureusement, tous les navigateurs gèrent correctement la balise `meta`, qui fournit la même information, à condition que vous pensiez à la faire systématiquement figurer dans vos `head` :

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-  
15" />
```

La valeur initiale dépend d'ailleurs de la façon dont vous voulez « servir » votre contenu. Ici, c'est du HTML. On peut aussi opter pour le mode « XML », qui opère quelques petits changements à l'usage, ce qui affecte notamment le comportement du DOM. Le type MIME à utiliser alors est `application/xhtml+xml`. Choisir entre les deux et gérer les conséquences fait l'objet d'un débat incessant depuis de longues années...

XHTML, oui mais lequel ?

La version actuelle est XHTML 1.1, qui est toutefois encore loin d'être utilisée partout. Cette version n'a plus les variantes Transitional et Loose de la version 1.0 : elle considère qu'on écrit désormais du XHTML forcément strict (même si on conçoit des cadres) !

La version 1.1 date déjà : elle fut publiée le 31 mai 2001, soit à peine 16 mois après la publication initiale de la version 1.0. Fait plus marquant encore : la dernière révision de XHTML 1.0 date du 1^{er} août 2002, elle est donc *plus récente* que XHTML 1.1...

Les différences avec XHTML 1.0, qui est de loin la variante la plus utilisée ces dernières années, sont très minimes, même pour une sous-version : l'attribut universel `lang` a été remplacé par `xml:lang` à des fins de compatibilité avec la modularisation en cours de XHTML, et une série d'éléments dits « ruby » a été ajoutée, qui permet de fournir des petits blocs de texte dans la marge, annotant le corps de texte principal.

Le gros du travail sur XHTML se concentre aujourd'hui sur XHTML 2.0, mais il s'agit là d'un des nombreux symptômes d'un malaise dans le W3C. En effet, l'immense majorité des ténors du Web ont vivement critiqué cette spécification, et estiment, pour résumer, que leurs auteurs se fourvoient.

Indiquons simplement qu'elle est énorme, divisée en pas moins de 26 modules (la tendance au W3C étant à la modularisation ces dernières années), certains étant minuscules (attributs noyau), d'autres énormes (XForms).

Les développeurs web et leurs spécialistes ont le sentiment que le comité n'a jamais eu à rédiger véritablement un site web, et travaille dans sa tour d'ivoire pour produire une recommandation que personne n'utilisera jamais...

Le balisage sémantique

C'est un concept très simple : il s'agit d'utiliser une balise pour son sens et non pas pour son aspect par défaut.

Encore aujourd'hui, vous trouverez nombre de développeurs web qui vous diront que « `h1`, c'est pour écrire en gros », ou que « `blockquote` sert à décaler vers la droite ». Ce genre de perception remonte aux premiers navigateurs, vers 1994. Autant dire, à l'échelle du Web, que ça remonte au Jurassique.

L'aspect par défaut d'une balise est sans aucune importance. Je dis bien : aucune. D'ailleurs, il n'existe aucun standard pour ces aspects par défaut (juste une suggestion en annexe D de la recommandation CSS 2.1). Vous ne trouverez pas un seul document formel disant à quoi doit ressembler une balise de titre, ni même `strong` ou `em`.

Bien sûr, on a longtemps eu des balises dédiées à l'aspect, par exemple `font`, `b`, `i`, `u` et `s`. Depuis HTML 4.01 (depuis 7 ans), seules les balises `b` et `i` sont encore autorisées, en raison de certains usages fréquents qui restent plus simples, et plus propres finalement, que de coller un `...` à la place.

Il est temps d'apprendre le véritable sens de ces balises. Ce faisant, vous découvrirez certaines balises et attributs dont vous n'aviez jamais entendu parler. Par exemple, connaissez-vous les balises `abbr`, `acronym`, `caption`, `cite`, `del`, `fieldset`, `ins`, `kbd`, `label`, `legend`, `q`, `samp`, `tbody`, `tfoot`, `th`, `thead` et `var`? Je suis sûr qu'au moins certaines ne vous disent absolument rien. Et que dire des attributs `accesskey`, `axis`, `datetime`, `dir`, `for`, `lang`, `longdesc`, `scope` et `summary`? Tout cela est pourtant lié à la sémantique et à l'accessibilité.

Ce n'est vraiment pas difficile de changer votre façon d'écrire du HTML. Commencez par aller découvrir les balises et attachez-vous à bien comprendre leur sens et leurs contextes autorisés d'utilisation. Lisez bien toute la description de la balise et de ses attributs. Fuyez toute balise et tout attribut dépréciés. La recommandation W3C est là pour ça, elle est claire, et HTML 4.01 dispose même d'une version française de bonne qualité (voir en fin de cette annexe).

Les balises XHTML 1 Strict par catégorie

Cette section liste l'ensemble des balises autorisées en HTML 4.01 et donc en XHTML 1 (1.0/1.1, à l'exception du module Ruby de XHTML 1.1, trop peu implémenté, et des variantes Transitional et Loose de 1.0). Les balises dépréciées sont rapidement listées, dans une catégorie à part.

Dans les descriptions, vous trouverez une explication courte du sens et du contexte d'utilisation éventuel pour la balise, ainsi que quelques remarques, mises en garde, et surtout invitations à aller explorer certains attributs trop souvent ignorés ou mal employés.

Il ne s'agit pas de vous apprendre le HTML, pas plus que l'annexe B n'a pour ambition de vous apprendre CSS. Il y a d'excellents livres et sites pour cela ; vous en trouverez quelques-uns dans les sections « Pour aller plus loin... » de ces annexes. Il s'agit plutôt ici de vous rappeler les fondamentaux, et de vous ouvrir les yeux sur le reste.

Balises structurelles

Les balises structurelles sont celles qui décrivent la structure du document : en-têtes, corps, regroupement d'éléments en ligne ou de type bloc, tableaux...

Tableau A-1 Balises structurelles de XHTML 1

Balise	Description
body	Corps du document. Ne peut contenir que des éléments de type bloc, <code>script</code> , <code>ins</code> ou <code>del</code> . Les attributs <code>background</code> , <code>text</code> , <code>link</code> , <code>vlink</code> et <code>alink</code> ont été dépréciés en faveur des CSS.
col	Décrit le format d'une colonne de tableau. Présente dans <code>table</code> ou <code>colgroup</code> .
colgroup	Alternative de description des formats d'un groupe de colonnes. Présente dans <code>table</code> .
div	Conteneur de type bloc utilisé pour grouper d'autres éléments à des fins de CSS ou de script. Ne sombrez pas inutilement toutefois dans la <i>divitis</i> (http://fr.wikipedia.org/Divitis) !
head	En-têtes du document : titre, métadonnées, liens vers des feuilles de styles, scripts et documents connexes.
html	L'élément racine de tout document (X)HTML. Pour être impeccable, on renseigne ses attributs <code>xmlns</code> et <code>xml:lang</code> , comme indiqué dans la section 3.1.1 de la recommandation pour XHTML 1.0.
script	Permet de charger un script séparé (usage correct, dans le cadre de <code>head</code>) ou de définir un script interne. Ce dernier usage lui vaut sa place ici, mais il va à l'encontre des principes de l' <i>unobtrusive JavaScript</i> , décrits au chapitre 2. Voir également le tableau A-3.
span	Conteneur servant à regrouper des éléments de type <i>inline</i> à des fins de CSS ou de script.
table	Tableau de données. Ne jamais utiliser pour obtenir une mise en page ! Connaissez-vous son attribut <code>summary</code> ?
tbody	Conteneur de tout ou partie des lignes du corps d'un tableau, présent dans <code>table</code> après d'éventuels <code>thead</code> et <code>tfoot</code> . Oubliez les <code>tr</code> directement dans <code>table</code> , ce n'est pas soigné ! Lorsqu'un tableau contient plusieurs sous-sections logiques, il est bon de consacrer un <code>tbody</code> à chacune.
td	Cellule de tableau contenant une donnée (<i>Table Data</i>). Les attributs <code>abbr</code> , <code>headers</code> et <code>scope</code> font des merveilles pour l'accessibilité à l'intention des logiciels de lecture d'écran ou des personnes souffrant d'un handicap cognitif.
tfoot	Pied de tableau. Présente dans <code>table</code> après un éventuel <code>thead</code> , mais avant <code>tbody</code> . Censée être répétée en bas de chaque portion visible du tableau quand celui-ci est imprimé sur plusieurs pages, par exemple.
th	Cellule de tableau contenant un titre de ligne ou de colonne (<i>Table Heading</i>). Même remarque que pour <code>td</code> .
thead	En-tête de tableau. Présente dans <code>table</code> comme premier élément fils (exception faite de <code>caption</code>). Censée être répétée en haut de chaque portion visible du tableau quand celui-ci est imprimé sur plusieurs pages, par exemple.
tr	Ligne de cellules dans une portion de tableau (<i>Table Row</i>). Peut contenir des éléments <code>th</code> ou <code>td</code> .

Balises sémantiques

Les balises sémantiques forment la plus grande catégorie, pour la simple raison que HTML est conçu pour indiquer au mieux la signification des portions de texte qui forment son contenu. Si ce tableau ne vous fait pas découvrir un seul élément, chapeau bas !

Tableau A-2 Balises sémantiques de XHTML 1

Balise	Description
abbr	Indique une abréviation. Si on cherche l'exactitude, une abréviation est soit la version raccourcie d'un mot (par exemple « Mass. » pour « Massachusetts »), soit une série d'initiales n'étant pas conçue pour être prononcée, mais plutôt épelée (WWW, URI, HTTP, SNCF...).
acronym	Indique un acronyme, c'est-à-dire une série d'initiales conçue pour être prononcée (Wysiwyg, Radar, Adullact...).
blockquote	Bloc de citation, approprié pour une citation longue. L'attribut cite permet de fournir l'URI de la source. Voir aussi q.
caption	Titre d'un tableau, préférable de loin à une première tr dans thead avec un th doté d'un éventuel attribut colspan. Élément fils immédiat de table. Généralement affiché par défaut au-dessus du tableau (réglable par CSS), centré sur la largeur de celui-ci.
cite	Référence à une source externe (citation au sens de « citer ses sources »). Voir aussi blockquote et q.
code	Fragment de code informatique (dans un langage quelconque). Voir aussi kbd, samp et var.
dd	Corps d'une définition associée au dernier terme (dt) la précédent dans une liste de définition (dl). Un même terme peut avoir plusieurs définitions successives, ou plusieurs fragments de définition, sous forme de plusieurs éléments dd.
del	Indique une excision (retrait) de contenu dans le cadre d'une révision du document. Un des éléments les plus sous-utilisés de HTML. L'attribut datetime permet de dater la modification, et l'attribut cite peut référencer l'URI d'un document détaillant ses motivations. L'affichage par défaut est explicite, en laissant son contenu visible mais barré. Voir aussi ins.
dfn	La balise la plus mal spécifiée de tout HTML. Censée encadrer la première occurrence d'un terme, à valeur de définition. Mais la recommandation n'explique pas comment isoler le mot dans sa définition...
dl	Liste de définitions. Contient une série de termes (dt) suivis d'une ou plusieurs définitions (dd). La sémantique autorisée est un peu plus large que les définitions à proprement parler, de sorte qu'on peut s'en servir pour un glossaire, un menu, voire même des dialogues dans un scénario.
dt	Terme à définir, dans le cadre d'une liste dl. Suivie d'une ou plusieurs définitions dd.
em	Emphase simple, plus légère que strong. Dans un texte, correspond souvent à l'effet produit par la mise en italique.
h1...h6	Titres à niveaux hiérarchiques, h1 étant le premier niveau (le plus haut), h6 le dernier (on en a rarement besoin, ou alors votre page est un immense pavé). Les moteurs de recherche leur accordent en général un « poids » proportionnel.
img	Insère une image. Devrait toujours avoir un attribut alt, vide uniquement si l'image est purement décorative (pour ne pas gêner la représentation textuelle). Les attributs longdesc et title ont aussi un rôle à jouer dans l'accessibilité.
ins	Symétrique à del : indique un ajout de contenu dans le cadre d'une révision du document. Un des éléments les plus sous-utilisés de HTML. L'attribut datetime permet de dater la modification, et l'attribut cite peut référencer l'URI d'un document détaillant ses motivations.
kbd	Indique un texte saisi par l'utilisateur dans le cadre d'une manipulation. Voir aussi samp.

Tableau A–2 Balises sémantiques de XHTML 1 (suite)

Balise	Description
label	Indique un libellé de champ dans un formulaire. Figure aussi au tableau A-6 pour l'aspect formulaire. Ses attributs <code>accesskey</code> et <code>for</code> sont primordiaux, car certains types de champs ne peuvent définir leur propre <code>accesskey</code> . Attention : <code>for</code> référence un <code>id</code> , pas un <code>name</code> ! Peut se présenter sous deux formes, mais on préfère qu'il n'encadre que le libellé lui-même, et pas le champ : cela ouvre davantage de possibilités tant pour les CSS que pour les scripts.
li	Élément de liste (<code>ul</code> ou <code>ol</code>). Pensez à bien les fermer en XHTML !
ol	Liste ordonnée (séquence croissante de numéros ou lettres). Tous les attributs de séquencement (<code>start</code> , <code>value</code>) et de format (<code>type</code> , <code>compact</code>) ont été dépréciés au profit des CSS, plus puissantes d'ailleurs.
p	Définit un paragraphe, ce qui est en soi un type sémantique de contenu. Pensez à bien les fermer en XHTML !
q	Citation courte. L'attribut <code>cite</code> permet de référencer l'URI de la source. Voir aussi <code>blockquote</code> .
samp	Exemple d'un affichage ou d'une sortie imprimée par un programme (affichage en console, etc.). Voir aussi <code>kbd</code> .
strong	Emphase forte, plus appuyée que <code>em</code> . Dans un texte, correspond à l'effet produit par la mise en gras.
sub	Texte en indice, généralement dans une formule (<i>subscript</i>).
sup	Texte en exposant, généralement dans une formule ou pour une note de bas de page (<i>superscript</i>).
title	Titre de la page, présente dans <code>head</code> . Contient souvent au moins l'équivalent de l'éventuel élément <code>h1</code> présent dans le corps de la page, mais avec plus de contexte. Son contenu constitue souvent le titre de la fenêtre ou de l'onglet du navigateur.
ul	Liste non ordonnée (à puces). Les attributs <code>type</code> et <code>compact</code> ont été dépréciés au profit des CSS, plus puissantes d'ailleurs. Idéal pour représenter sémantiquement un menu, les CSS pouvant le transformer à volonté...
var	Indique une variable ou un argument de programme. Voir aussi <code>code</code> et <code>kbd</code> .

Balises de liaison

Il s'agit de balises établissant des liens vers d'autres ressources ou documents.

Tableau A–3 Balises de liaison de XHTML 1

Balise	Description
a	La balise de lien par excellence. Attention : son attribut <code>target</code> est déprécié depuis déjà 7 ans, et l'attribut <code>name</code> n'est plus autorisé en XHTML : utilisez plutôt un <code>id</code> sur l'élément vers lequel vous souhaitez définir une URL de fragment. Côté sémantique, l'utilisation de l'attribut <code>hreflang</code> est en pleine explosion (notamment grâce aux sélecteurs CSS 2), l'attribut <code>rel</code> commence à faire surface grâce à des fonctionnalités proposées par Google, et l'attribut <code>type</code> peut être utile.
base	Modifie l'URL de base pour les URL relatives au sein du document. Présente dans <code>head</code> . Son attribut <code>target</code> permet, le cas échéant, de faire que tous les liens s'ouvrent dans un autre cadre que le cadre courant.

Tableau A-3 Balises de liaison de XHTML 1 (suite)

Balise	Description
link	Lien vers une ressource externe. Principal moyen de chargement de CSS. Ses attributs type et href sont incontournables, mais l'attribut rel est aussi très utile, tant pour proposer plusieurs feuilles de styles alternatives que pour définir des hiérarchies ou réseaux de pages. Connaissez-vous ses valeurs start, next, prev, contents, chapter ou encore section ? Il y en a bien d'autres : http://www.w3.org/TR/html401/types.html#type-links . Enfin, l'attribut hreflang existe ici aussi, ce qui est sémantiquement intéressant.
script	Chargement de scripts externes depuis le head. Seule utilisation correcte, contrairement aux fragments de scripts intégrés au contenu de la page (à l'exception des retours Ajax, comme cela est montré dans plusieurs chapitres). Attention : l'attribut language est déprécié depuis 7 ans, au profit de l'attribut type, qui vaut donc le plus souvent text/javascript.

Balises de métadonnées

Tableau A-4 Balises de métadonnées de XHTML 1

Balise	Description
address	Encadre les informations de contact des responsables de la page (liens mailto:, paragraphes avec une adresse postale,etc.). Extrêmement peu utilisé. À tort ?
bdo	Court-circuite l'algorithme déterminant le sens du texte (gauche à droite ou droite à gauche) sur la base des attributs dir des éléments contenant celui-ci (<i>Bi-Directional Override</i>). Son attribut dir redéfinit la direction par défaut de son contenu textuel, tandis que son attribut lang (ou xml:lang en XHTML 1.1) permet d'en changer aussi la langue indiquée. J'avoue ne pas percevoir l'intérêt par rapport à span, qui n'a pas plus de valeur sémantique et peut faire la même chose...
meta	L'élément de métadonnées par excellence. Figure dans head, autant de fois que nécessaire. Peut aussi bien remplacer ou compléter les en-têtes de réponse HTTP (attribut http-equiv, très utilisé notamment pour préciser le jeu de caractères grâce à Content-Type) que rajouter des métadonnées quelconques (attribut name). La valeur est dans l'attribut content. On l'utilise notamment pour remplir les pages de métadonnées appartenant à des ontologies (vocabulaires) reconnues, comme le Dublin Core (http://dublincore.org/documents/dcmi-terms/). Connaissez-vous son attribut scheme, plutôt avancé ?

Balises de présentation

Tableau A-5 Balises de présentation de XHTML 1

Balise	Description
area	Définit une zone cliquable sur une image à l'aide de ses attributs shape, coords et href. Ne négligez surtout pas ses attributs alt, tabindex et accesskey pour l'accessibilité ! Voir aussi map.
b	Mise en gras. Jugé plus soigné qu'un <code>...</code> en l'absence de connotation sémantique.

Tableau A-5 Balises de présentation de XHTML 1 (suite)

Balise	Description
big	Utilisation de la taille de texte supérieure. J'avoue ne pas en voir l'utilité par rapport aux CSS. Voir aussi small .
br	Fin de ligne, ou retour chariot si vous préférez (<i>line break</i>). Force le passage à la ligne dans un texte.
frame	Définition d'un cadre dans un ensemble de cadres. Je rappelle que les cadres sont à éviter le plus possible, car ils nuisent à l'accessibilité et à la navigation. Si vous en utilisez néanmoins, assurez-vous de bien définir l'attribut longdesc .
frameset	Définition d'un ensemble de cadres, qui précise notamment la disposition (horizontale ou verticale).
hr	Ligne horizontale de séparation (<i>horizontal rule</i>). Tous ses attributs spécifiques sont dépréciés au profit des CSS.
i	Mise en italique. Jugé plus soigné qu'un <code>...</code> en l'absence de connotation sémantique.
map	Conteneur descriptif pour les zones cliquables (area) d'une image (img ; je vous déconseille d'utiliser des zones cliquables sur un input type="image" ou un object , même si c'est autorisé). Son attribut name est référencé par l'attribut usemap de l'image. Attention à bien renseigner les attributs d'accessibilité des balises area .
noframes	Contenu alternatif pour un navigateur ne prenant pas en charge les cadres. Principalement utile pour les moteurs de recherche, qui n'iront pas dans les cadres. En revanche, tous les navigateurs répandus, même textuels (w3m a supplanté lynx), gèrent aujourd'hui les cadres.
object	Balise fourre-tout pour objets ne disposant pas d'une balise dédiée : applets Java (la balise applet est dépréciée depuis 7 ans), animations Flash ou Shockwave, et sur MSIE n'importe quel contrôle ActiveX (par exemple Windows Update). Essentiellement utilisé pour fournir des animations et des lecteurs audio ou vidéo au sein de la page.
param	À l'intérieur d'un object , les éléments param permettent de définir des paramètres nommés pour l'objet.
pre	Texte préformaté : il s'agit généralement de l'équivalent d'un <code><div style="white-space: pre; ></code> . On y laisse donc normalement les espacements tels quels, ainsi que les retours chariot : les lignes ne sont pas découpées pour satisfaire une contrainte de largeur. Très utilisée pour présenter du code source ou tout autre contenu textuel de nature informatique. La police de caractères par défaut est à chasse fixe (par exemple Courier New).
small	Utilisation de la taille de texte inférieure. J'avoue ne pas en voir l'utilité par rapport aux CSS. Voir aussi big .
style	Permet de fournir un fragment de script dans le corps du document. Cette utilisation est à proscrire, car elle est contraire à la séparation du contenu et de l'aspect. Ne pas confondre avec l'attribut style de la plupart des éléments, qui est parfois justifié dans certains contextes très délimités. Voir aussi link dans le tableau A-3.
tt	Utilisation d'une police à chasse fixe (teletype). À n'utiliser que si vous estimez que votre contenu n'est pas couvert sémantiquement par code , kbd , samp ou var .

Balises de formulaires et d'interaction

La plupart des champs de formulaires et boutons disposent d'attributs relatifs à l'ergonomie et l'accessibilité, qu'il vous faut bien connaître. Citons notamment `accesskey` (qu'il est préférable de laisser toutefois sur un `label` associé), `disabled`, `readonly` et `tabindex`.

Tableau A-6 Balises de formulaire et d'interaction de XHTML 1

Balise	Description
<code>button</code>	Variante peu utilisée (car souvent visuellement... bizarre) de <code><input type="button"... /></code> . La balise <code>button</code> est en fait un conteneur, dans lequel on peut placer tout type de contenu en ligne, à l'exception des balises de formulaire et de <code>a</code> . Elle est donc censée permettre un contenu de bouton plus riche que <code>input</code> . Personnellement, je ne m'en sers jamais.
<code>fieldset</code>	Groupe logique de champs dans un formulaire, doté le plus souvent d'un titre grâce à <code>legend</code> . Encore trop peu utilisé et pourtant très utile pour des formulaires un peu voire très riches en contenu. Vous verrez un exemple plus loin dans cette annexe.
<code>form</code>	Formulaire. Grammaticalement, il n'est pas possible de placer un champ de formulaire hors d'une balise <code>form</code> , quitte à ce qu'elle soit inutile par ailleurs. Les principaux attributs sont <code>method</code> et <code>action</code> . Si vous avez des champs de type fichier, l'attribut <code>enctype</code> est indispensable. Quant à <code>name</code> , il est déprécié depuis XHTML 1.0, au profit de <code>id</code> . Par ailleurs, la DTD exige que <code>form</code> ne contienne que des éléments de type <code>bloc</code> ou <code>script</code> , hormis <code>form</code> bien sûr. Conséquence : vos champs de formulaire doivent apparaître dans des conteneurs de type <code>bloc</code> , souvent <code>p</code> ou <code>fieldset</code> .
<code>input</code>	La principale balise de description de champ. Son attribut <code>type</code> , quoique doté d'une valeur par défaut (<code>text</code>), devrait toujours être précisé, ainsi que <code>name</code> (sauf pour le type <code>submit</code> , à moins qu'il y en ait plusieurs) et <code>tabindex</code> . Pensez aussi, quand c'est pertinent, aux attributs <code>accept</code> , <code>checked</code> , <code>disabled</code> , <code>maxlength</code> , <code>readonly</code> et <code>value</code> . Quant à <code>size</code> , préférez utiliser une règle CSS.
<code>label</code>	Libellé d'un champ de formulaire. Signalé au tableau A-2 pour son rôle sémantique, ce pour quoi je vous encourage bien entendu à fournir un libellé pour tout champ n'en ayant pas déjà un (donc, tous sauf les boutons). Pensez aussi à associer explicitement le <code>label</code> à son champ à l'aide d'un <code>id</code> sur le champ et du <code>for</code> correspondant sur le <code>label</code> . <code>select</code> ne disposant pas d'attribut <code>accesskey</code> , <code>label</code> permet de lui en associer un ; d'une manière générale, mettez les <code>accesskey</code> sur les libellés quand ils existent.
<code>legend</code>	Titre d'un groupe logique de champs et libellés dans un formulaire. Généralement le premier élément fils d'un <code>fieldset</code> .
<code>optgroup</code>	Dans un <code>select</code> , définit un groupe d'options possibles, ce qui est pratique quand il y en a beaucoup. On utilise l'attribut <code>label</code> pour nommer le groupe, et on peut même désactiver juste ce groupe avec <code>disabled</code> . On ne peut pas imbriquer les éléments <code>optgroup</code> : la hiérarchie n'a qu'un niveau. Suivant le navigateur, l'aspect par défaut varie mais un style peut bien entendu être appliqué.
<code>option</code>	Option sélectionnable dans un <code>select</code> . Peut figurer directement sous le <code>select</code> ou dans un <code>optgroup</code> . L'attribut <code>selected</code> parle de lui-même... Désactivable individuellement avec l'attribut <code>disabled</code> . Il est préférable de toujours préciser l'attribut <code>value</code> , le comportement normatif (utilisation du contenu textuel comme valeur par défaut) n'étant pas respecté au niveau DOM par MSIE.

Tableau A–6 Balises de formulaire et d'interaction de XHTML 1 (suite)

Balise	Description
script	Permet l'inclusion de fragments de script directement dans le corps du document, principalement pour manipuler un fragment DOM juste après sa création, sans attendre le chargement complet de la page. C'est une pratique qui va plutôt à l'encontre de l' <i>unobtrusive JavaScript</i> , aussi je la déconseille fortement. Là encore, il existe quelques cas, notamment la récupération de fragments XHTML + JS en Ajax, où cette utilisation est toutefois acceptable.
select	Liste d'options, soit déroulante (attribut size de valeur 1), soit dépliée mais avec un nombre d'options visibles fixe (valeur de l'attribut size). Peut contenir des balises option et optgroup. L'attribut multiple, s'il est fourni, indique que l'utilisateur peut sélectionner plusieurs éléments. Attention : il est ergonomiquement incompatible avec le mode déroulant.
textarea	Zone de saisie de texte multiligne. Utilise une balise fermante. Attention : tout le contenu entre la fin de la balise ouvrante et le début de la fermante fait partie de la valeur, espacements et retours chariot compris ! Les attributs importants pour l'aspect par défaut sont cols et rows. MSIE y ajoute une pléthora d'extensions, comme pour tous les champs de formulaire, mais la plus fréquemment rencontrée est l'attribut wrap. Évitez tous ces ajouts non standards.

Balises dépréciées

Voilà déjà 7 ans (HTML 4.01, décembre 1999) que les balises dépréciées n'ont plus droit de cité. Et pourtant, on les rencontre encore souvent... hélas ! Évitez donc de vous attirer les foudres des navigateurs modernes et de vos pairs.

Tableau A–7 Balises dépréciées de XHTML 1

Balise	Description
applet	Permettait l'inclusion d'une applet Java. Remplacée par object.
basefont	Permettait de préciser les valeurs par défaut pour les attributs des balises font.
center	Centrait tout son contenu. Remplacée par les propriétés CSS text-align et margin (valeur spéciale auto). Attention : l'attribut align est également déprécié pour tous les éléments !
dir	Devait permettre de créer des listes à colonnes multiples, comme un annuaire (directory). N'a jamais été vraiment prise en charge. Voir aussi menu.
font	Remplacée par les CSS, bien plus puissantes d'ailleurs.
iframe	Sorte de cadre en ligne, qui était abondamment utilisé (et, hélas, l'est encore) pour réaliser des échanges en arrière-plan avec le serveur (façon Ajax), ce qui était au moins autant utilisé à des fins légitimes qu'à d'autres bien plus dangereuses. Pas véritablement dépréciée, mais qui ne figurait déjà plus que dans la variante Loose de HTML 4.01, la moins soignée.
isindex	Très vieille variante de champ de saisie textuelle à ligne unique.
menu	Devait permettre de créer des listes à colonne unique. N'a jamais été vraiment prise en charge. Voir aussi dir.
s	Synonyme de strike.

Tableau A–7 Balises dépréciées de XHTML 1 (suite)

Balise	Description
strike	Barrait un contenu. Remplacée par <code>del</code> quand la sémantique convient, par la propriété CSS <code>text-decoration</code> sinon.
u	Soulignait le texte. Remplacée par la propriété CSS <code>text-decoration</code> .

Attributs incontournables

Pour finir, voici quelques attributs incontournables, dits « noyau », identifiables dans la DTD sous l'entité `%coreattrs`. Tous les éléments en disposent.

Tableau A–8 Attributs incontournables de XHTML 1

Attribut	Description
<code>class</code>	Contient une ou plusieurs classe(s) CSS. Le saviez-vous ? Il est tout à fait possible de fournir plusieurs classes à un même élément, qui cumule alors les applications de règles. Cela est très pratique, et évite d'avoir à dupliquer nombre de règles dans la feuille de styles. Les classes sont séparées dans la valeur de l'attribut par des espaces.
<code>dir</code>	Indique la direction d'écriture du contenu (<code>ltr</code> ou <code>rtl</code> , pour gauche à droite ou droite à gauche). Voir aussi <code>lang</code> .
<code>id</code>	De loin l'attribut le plus utilisé, avant même <code>class</code> , dans un document soigné et sémantique ! Contient un identifiant, unique à travers tout le document, grâce auquel tant les CSS que les scripts pourront référencer l'élément en un clin d'œil. Le saviez-vous ? Un ID valide doit commencer par une lettre non accentuée et peut ensuite contenir des chiffres arabes et des traits de soulignement (<code>_</code>), mais aussi des tirets (<code>-</code>), des points (<code>.</code>) et des deux-points (<code>:</code>) ! Ces deux dernières possibilités peuvent toutefois poser un problème dans la syntaxe des règles CSS, qui ont alors besoin d'encadrer les ID par des apostrophes (<code>'</code>) pour s'y retrouver.
<code>lang</code>	Langue du contenu. Contient un code de langue de la RFC 1766, par exemple <code>fr-FR</code> . Voir aussi <code>dir</code> .
<code>style</code>	Style en ligne appliqué à l'élément. Généralement mal vu, car enfreint la séparation entre contenu et aspect. Parfois nécessaire cependant, en particulier pour masquer initialement un contenu avec <code>display: none</code> dans le cadre des bibliothèques Prototype et donc <code>script.aculo.us</code> (voir chapitres 4, 7 et 8).
<code>title</code>	Fournit une information descriptive ou complémentaire sur l'élément. L'information est généralement affichée dans une infobulle si le curseur de la souris reste un instant sur l'élément. Fondamental pour des balises comme <code>abbr</code> et <code>acronym</code> , il est aussi utile pour préciser la destination d'un lien ou nommer les feuilles de styles alternatives d'un document. Il a enfin de nombreux usages dans le cadre de l'accessibilité.

Besoins fréquents, solutions concrètes

À présent que nous avons fait le tour des balises disponibles en XHTML 1 Strict, il est temps d'en illustrer l'utilisation la plus correcte possible dans le cadre de certains besoins récurrents.

Un formulaire complexe mais impeccable

Prenons un formulaire sur un site communautaire, servant à éditer des informations de profil. Les concepteurs du site ont décidé que la totalité des informations étaient présentées dans un même formulaire. Ces informations sont réparties en 5 groupes logiques :

- 1 connexion (identifiant, mot de passe) ;
- 2 identité (civilité, nom, prénom, surnom) ;
- 3 détails personnels (adresse courriel, identifiants de messagerie instantanée, site personnel, petit descriptif en texte libre, photo) ;
- 4 centres d'intérêt (série de cases à cocher) ;
- 5 conditions de fonctionnement (CGU, divulgation d'informations à des partenaires, type d'abonnement à la lettre de diffusion du site).

Le formulaire doit être le plus lisible possible, pour tout type de public. On veillera donc à éviter une mise en page tableau, pour lui préférer l'emploi de CSS adaptables au type de navigateur (petit ou grand écran, graphique ou texte, lecteur d'écran ou plage braille, etc.) et aux besoins de l'utilisateur (navigation intelligente à l'aide de la touche Tab et de touches de raccourci, libellés associés aux champs, etc.).

La photo peut être téléchargée à l'aide d'un champ de type fichier. Ce champ doit filtrer les possibilités pour ne retenir que les formats d'image pris en charge par le site, pour lequel on a décidé de se limiter à JPEG, PNG et GIF. Une limite de taille de fichier sera gérée côté serveur, et l'image y sera de toutes façons redimensionnée.

Évidemment, toute validation ou complément fonctionnel JavaScript sera réalisé de façon non intrusive, depuis un script séparé. Aucun gestionnaire d'événement ne sera présent dans le code, et un envoi sans activation JavaScript sera possible (avec un champ de type `submit` ou `image`).

Voici le code HTML répondant à ce besoin (vous le retrouverez dans l'archive de codes source pour ce livre, disponible sur le site des éditions Eyrolles).

Listing A-1 Notre première section, dédiée aux informations de connexion

```
<form id="profile" enctype="multipart/form-data"
      method="post" action="/profile/update">
```

```
<fieldset id="credentials">
    <legend>Connexion</legend>
    <p>
        <label for="edtReqLogin" accesskey="E">Identifiant</label>
        <input type="text" id="edtReqLogin" name="login"
               ↪ value="tdd" tabindex="1" />
    </p>
    <p>
        <label for="edtPassword" accesskey="A">Nouveau mot de passe</label>
        <input type="password" id="edtPassword" name="password"
               ↪ tabindex="2" />
    </p>
    <p>
        <label for="edtConfirm" accesskey="C">Confirmation</label>
        <input type="password" id="edtConfirm" name="confirm"
               ↪ tabindex="3" />
    </p>
</fieldset>
```

- Notez l'attribut `enctype` du formulaire `form`, en prévision du champ de téléchargement de fichier du listing A-3.
- Chaque champ fait l'objet d'une valeur à progression logique pour `tabindex`.
- Le mot de passe étant saisi sans confirmation visuelle, on ajoute un champ de confirmation pour détecter une saisie erronée.
- Tous les libellés définissent une `accesskey` pour leur champ.

Voyons à présent la deuxième section du formulaire.

Listing A-2 Informations sur l'identité de l'utilisateur

```
<fieldset id="identity">
    <legend>Identité</legend>
    <p>
        <label for="cbxTitle" accesskey="V">Vous êtes</label>
        <span id="basicIdentity">
            <select id="cbxTitle" name="title" size="1" tabindex="4">
                <option value="1" selected="selected">M.</option>
                <option value="2">Mlle</option>
                <option value="3">Mme</option>
            </select>
            <input type="text" id="edtReqFirstName" name="firstName"
                   ↪ accesskey="P" value="Christophe" tabindex="5" />
            <input type="text" id="edtReqLastName" name="lastName"
                   ↪ accesskey="N" value="Porteneuve" tabindex="6" />
        </span>
    </p>
```

```

<p>
    <label for="edtNickname" accesskey="S">Surnom</label>
    <input type="text" id="edtNickname" name="nickname"
           ↪ value="TDD" tabindex="7" />
</p>
</fieldset>

```

- Afin de gagner en place et en ergonomie, nous plaçons les champs composant l'identité globale (civilité, prénom, nom) sur une même ligne. Pressentant un besoin de groupement pour la CSS, on encadre les trois par un `span`.
- Le `select`, en `size 1`, est une liste déroulante. L'élément retenu par défaut est marqué par un attribut bascule `selected`. Notez sa valeur, seule autorisée par la DTD.

La section sur les détails personnels est de loin la plus riche.

Listing A-3 Détails personnels

```

<fieldset id="details">
    <legend>Détails personnels</legend>
    <p>
        <label for="edtReqEmail" accesskey="0">Courriel</label>
        <input type="text" id="edtReqEmail" name="email"
               ↪ value="tdd@example.com" tabindex="8" />
    </p>
    <p>
        <label><abbr title="Instant Messaging,
                   ↪ Messagerie instantanée">IM</abbr></label>
        <span id="imIDs">
            <label for="edtMSN" accesskey="1">MSN</label>
            <input type="text" id="edtMSN" name="im[msn]"
                   ↪ value="tdd@example.com" tabindex="9" title="Alt+1" />
            <label for="edtICQ" accesskey="2">ICQ</label>
            <input type="text" id="edtICQ" name="im[icq]"
                   ↪ value="123456789" tabindex="10" title="Alt+2" />
            <label for="edtJabber" accesskey="3">Jabber</label>
            <input type="text" id="edtJabber" name="im[jabber]"
                   ↪ value="tdd@example.com" tabindex="11" title="Alt+3" />
        </span>
    </p>
    <p>
        <label for="edtHomePage" accesskey="T">Site personnel</label>
        <input type="text" id="edtHomePage" name="homePage"
               ↪ value="http://www.example.com" tabindex="12" />
    </p>
    <p id="detailsContainer">
        <label for="memDetails" accesskey="D">Détails</label>
        <textarea id="memDetails" name="details" cols="40" rows="3"
                  ↪ tabindex="13"></textarea>
    </p>

```

```
<p>
  <label for="filPhoto" accesskey="H">Photo</label>
  <span id="photo">
    <input type="file" id="filPhoto" name="photo" tabindex="14"
           ↪ accept="image/jpeg,image/gif,image/png" />
    <span class="note">(JPEG, PNG ou GIF, 150ko maximum)</span>
  </span>
</p>
</fieldset>
```

- Notez l'explication de l'abréviation IM, à l'aide d'abbr et de son attribut title. La CSS incitera l'utilisateur à amener sa souris sur le terme et à l'y laisser un instant pour obtenir l'explication.
- Là aussi, quelques éléments sur la même ligne sont regroupés dans un span en prévision des CSS.
- Remarquez comme les champs de saisie d'identifiants IM précisent leur touche de raccourci avec l'attribut title : le caractère actif ne figurant pas dans leur libellé, il ne pourra pas être mis en évidence par un script. Il faut tenter de fournir l'information autrement, et title produit généralement une infobulle.
- Remarquez que textarea est un élément non vide : il faut utiliser une balise fermante. Les suggestions de taille proposées par les attributs cols et rows seront remplacées par celles de la CSS quand celle-ci est active.
- Exemple de champ de type file. Il n'est pas possible de spécifier une valeur dans le HTML. Notez l'attribut accept.

La section 4, celle des centres d'intérêt, est plus simple. On suppose que la liste de ceux-ci est dynamiquement extraite de la base de données, ce qui empêche l'utilisation de touches de raccourci, car on ne connaît pas forcément tous les libellés affichés.

Listing A-4 Centres d'intérêt

```
<fieldset id="interests">
  <legend>Centres d'intérêt</legend>
  <ul>
    <li>
      <input type="checkbox" id="chkInterests_1"
             ↪ name="interests" value="1" />
      <label for="chkInterests_1">Jeux vidéo</label>
    </li>
    <li>
      <input type="checkbox" id="chkInterests_2"
             ↪ name="interests" value="2" />
      <label for="chkInterests_2">Mode</label>
    </li>
  </ul>
</fieldset>
```

```

<li>
    <input type="checkbox" id="chkInterests_3"
           ↪ name="interests" value="3" />
    <label for="chkInterests_3">Gadgets</label>
</li>
</ul>
</fieldset>

```

Ici, une liste non ordonnée est sémantiquement parfaite pour représenter cette liste de choix individuels. La CSS fera disparaître les puces et ajustera les positions. Pour faciliter le traitement côté serveur, on donne à tous les champs le même nom, mais une valeur différente (ici probablement l'ID de l'information dans la base), utilisée pour composer l'id HTML et ainsi associer le libellé à la case. Cette association permet à l'utilisateur de basculer l'état de la case en cliquant sur le libellé lui-même, ce qui est plus accessible et similaire aux interfaces classiques. Côté serveur, on recevra le champ une fois par case cochée, avec à chaque fois la bonne valeur.

Enfin, voici la dernière section, le bouton d'envoi et la clôture du formulaire.

Listing A-5 Conditions de fonctionnement

```

<fieldset id="account-behavior">
    <legend>Conditions de fonctionnement</legend>
    <ul>
        <li>
            <input type="checkbox" id="chkCGU" name="cgu"
                   ↪ value="yes" checked="checked" />
            <label for="chkCGU" accesskey="U">J'accepte les
            conditions générales d'utilisation.</label>
        </li>
        <li>
            <input type="checkbox" id="chkSpreadData"
                   ↪ name="spreadData" value="yes" />
            <label for="chkSpreadData" accesskey="J">Je consens à ce
            que mes informations personnelles soient divulguées à des partenaires.
        </label>
        </li>
    </ul>
    <h2>La <span lang="en">newsletter</span> du site</h2>
    <ul>
        <li>
            <input type="radio" id="rbtNoSub" name="newsletter"
                   ↪ value="no" checked="checked" />
            <label for="rbtNoSub" accesskey="R">Je ne désire pas la
            recevoir.</label>
        </li>
    
```

```
<li>
    <input type="radio" id="rbtWeekly"
        ↳ name="newsletter" value="weekly" />
        <label for="rbtWeekly" accesskey="B">Je désire la
recevoir hebdomadairement.</label>
    </li>
    <li>
        <input type="radio" id="rbtMonthly"
            ↳ name="newsletter" value="monthly" />
            <label for="rbtMonthly" accesskey="L">Je désire la
recevoir mensuellement.</label>
    </li>
</ul>
</fieldset>
<p class="submit"><input type="submit" id="btnSubmit" value="Mettre
à jour mon profil" accesskey="M" title="Alt+M" /></p>
</form>
```

- Pour une case à cocher isolée, on précise la valeur afin d'éviter de dépendre du navigateur pour la valeur par défaut (suivant les cas on obtient on, 1, yes...), ce qui compliquerait le test côté serveur. Tester simplement la présence du champ dans la requête ne serait pas très robuste.
- Remarquez l'attribut bascule checked et sa seule valeur autorisée.
- Notez la précision de la langue pour le terme « newsletter », qui permettra notamment aux lecteurs d'écran de le prononcer correctement à l'intention des non-voyants ou malvoyants. La CSS pourrait même basculer en italique de tels éléments, par exemple (sélecteurs d'attribut CSS 2, ou de langue CSS 3).
- Les boutons radio mutuellement exclusifs portent le même nom, mais des valeurs différentes. Le champ n'est envoyé qu'une fois, pour le bouton sélectionné.
- Le bouton d'envoi, champ de type submit, n'a pas de libellé externe : il précise donc son accesskey et la rend consultable avec title. Il est par ailleurs inutile de lui donner un attribut name, ce qui enverrait un champ au serveur alors qu'il n'y a pas plusieurs modes d'envoi parmi lesquels choisir...

Le fichier HTML complet pèse 5 Ko, et la CSS en ajoute 1,5, soit un total de 6,5 Ko. Croyez-moi sur parole, l'équivalent en HTML des années 1990 est beaucoup plus lourd, alors qu'il est aussi beaucoup plus rigide et incapable de s'adapter d'un mode de consultation à un autre.

La figure A-1 vous montre le résultat sans aucune feuille de styles.

Figure A-1
Notre formulaire sans aucune mise en page

The screenshot shows a Mozilla Firefox browser window with the title "Votre profil - Mozilla Firefox". The address bar displays "file:///home/tdd/perso/livres/pages_1". The main content area contains the following form fields:

- Connexion:** Identifiant
- Nouveau mot de passe:**
- Confirmation:**
- Identité:**
 - Vous êtes:
 - Surnom:
- Détails personnels:**
 - Courriel:
 - IM MSN: ICQ: Jabber/GTalk:
 - Site personnel:
 - Détails:
 - Photo: Parcourir... (JPEG, PNG ou GIF, 150ko maximum)
- Centres d'intérêts:**
 - Jeux vidéo
 - Mode
 - Gadgets
- Conditions de fonctionnement:**
 - J'accepte les conditions générales d'utilisation.
 - Je consens à ce que mes informations personnelles soient divulguées à des partenaires.
- La newsletter du site:**
 - Je ne désire pas la recevoir.
 - Je désire la recevoir hebdomadairement.
 - Je désire la recevoir mensuellement.

At the bottom of the form are two buttons: "Mettre à jour mon profil" and "Terminé".

Cela fait certes peur, mais ce n'est pas grave : le contenu de la page a du sens, *beaucoup de sens*. La figure A-2 montre le formulaire, avec une feuille de styles et l'application d'un petit script de décoration automatique de libellés, comme celui vu au chapitre 3 dans la section « Décoration automatique de labels » :

Quand je vous disais que les CSS font des miracles...

Il est par ailleurs possible d'y ajouter un script non intrusif de validation automatique de formulaire, comme vu au chapitre 3, dans la section « Validation automatique de formulaires ». Vous trouverez aussi en fin d'annexe, dans la section « Pour aller plus loin... », l'URL d'un article OpenWeb dédié à la conception de formulaires sémantiques.

Figure A-2

Le même formulaire, mis en forme par CSS et un script complémentaire

Votre profil - Mozilla Firefox

Fichier Edition Affichage Aller à Marque-pages Outils Aide

Désactiver Cookies CSS Form Images Information Divers Entourer

Votre profil

CONNEXION

Identifiant* tdd

Nouveau mot de passe

Confirmation

IDENTITÉ

Vous êtes M. Christophe Porteneuve

Surnom TDD

DÉTAILS PERSONNELS

Courriel* tdd@example.com

IM MSN tdd@example.com ICQ 123456789 Jabber tdd@example.com

Site personnel http://www.example.com

Détails

Photo Parcourir... (JPEG, PNG ou GIF, 150ko maximum)

CENTRES D'INTÉRÊTS

Jeux vidéo
 Mode
 Gadgets

CONDITIONS DE FONCTIONNEMENT

J'accepte les conditions générales d'utilisation.
 Je consens à ce que mes informations personnelles soient divulguées à des partenaires.

La newsletter du site

Je ne désire pas la recevoir.
 Je désire la recevoir hebdomadairement.
 Je désire la recevoir mensuellement.

Mettre à jour mon profil

Terminé

Un tableau de données à en-têtes groupés

Prenons maintenant l'exemple d'un tableau de données. La balise `table` et ses collègues : `thead`, `tbody`, `tfoot`, `tr`, `th` et `td`, ne sont pas à éviter comme la peste ! Simplement, elles servent à décrire des tableaux de données, pas à réaliser une mise en page aussi rigide (et lourde) qu'un parpaing.

Supposons que nous souhaitions réaliser un tableau représentant en abscisse des produits (classés par catégories) et en ordonnée des modes de livraison, eux-mêmes déclinés par zone géographique de livraison. On aimerait que les en-têtes du tableau persistent au-delà d'un saut de page à l'impression. On aimerait aussi qu'il soient sémantiques et un minimum accessibles aux utilisateurs de lecteurs d'écran.

Voici comment procéder...

Listing A-6 Un tableau bien structuré, sémantique et accessible

```
<table id="shipment" summary="Options de livraisons et leurs coûts">
    <caption>Coûts de livraison par mode et par zone géographique</caption>
    <thead>
        <tr>
            <th rowspan="2" class="bodyHeading">Mode</th>
            <th colspan="3">T-shirts</th>
            <th colspan="4">Gadgets</th>
        </tr>
        <tr>
            <th>1-</th>
            <th>11-</th>
            <th>&gt; 50</th>
            <th>&lt; 1kg</th>
            <th>1-</th>
            <th>5-</th>
            <th>&gt; 10kg</th>
        </tr>
    </thead>
    <tbody>
        <tr><th colspan="8" class="zone">Europe</th></tr>
        <tr>
            <th>ColiPoste</th>
            <td>2€</td>
            <td>5€</td>
            <td>15€</td>
            <td>5€</td>
            <td>15€</td>
            <td>22€</td>
            <td>30€</td>
        </tr>
        <tr>
            <th>Colissimo</th>
            ...
            </tr>
        </tbody>
        <tbody>
            <tr><th colspan="8" class="zone">Amérique du Nord, Maghreb, Australie</th></tr>
            ...
            </tbody>
    </table>
```

- Notez l'attribut `summary`, qui résume le contenu de la table, et l'élément fils `caption`, qui lui fournit un titre explicite.

- Les en-têtes sont, logiquement, dans `thead`. Les cellules qui y figurent ne fournissent que des titres, ce sont donc des `th`.
- Les différents corps de données (ici un par zone géographique) sont des `tbody`. Chaque corps a ici un sous-titre dédié, réalisé avec une première ligne ne comportant qu'un `th` sur toute la largeur.
- Chaque ligne démarre par un titre, donc un `th`, puis n'a que des données, donc des `td`.

Le fichier HTML pèse 2 Ko, la CSS, quoique bien aérée, ne pèse que 610 octets. Voici l'aspect du tableau, doté de cette feuille de styles minimalisté :

Figure A-3

Notre tableau avec un style léger

Options de livraison

COÛTS DE LIVRAISON PAR MODE ET PAR ZONE GÉOGRAPHIQUE

Mode	T-shirts			Gadgets			
	1-10	11-50	> 50	< 1kg	1-5kg	5-10kg	> 10kg
Europe							
ColiPoste	2€	5€	15€	5€	15€	22€	30€
Colissimo	5€	10€	30€	12€	27€	36€	45€
Éclair	15€	20€	45€	20€	50€	70€	100€
Amérique du Nord, Maghreb, Australie							
ColiPoste	4€	10€	30€	10€	30€	44€	60€
Colissimo	10€	20€	60€	24€	54€	72€	90€
Éclair	30€	40€	90€	40€	100€	140€	200€

Terminé

Si on souhaite rendre maximale l'accessibilité, en particulier à destination des lecteurs d'écran (non-voyants, malvoyants) et des personnes souffrant d'un handicap cognitif, les éléments `th` et `td` disposent d'attributs `abbr`, `headers` et `scope` extrêmement utiles. Vous pourrez en apprendre davantage sur cet aspect et les tableaux en général ici : <http://pompage.net/pompe/autableau/>.

Un didacticiel technique

Pour finir, penchons-nous sur les balises sémantiques en ligne, c'est-à-dire conçues pour indiquer le sens d'un fragment de texte. Nous allons prendre l'exemple d'une documentation technique qui, après avoir référencé quelques sources et cité un fragment pertinent de l'une d'elles, fournit aux lecteurs un morceau de code source et les guide à travers quelques manipulations au clavier dans leur console.

Listing A-7 Une documentation au balisage hautement sémantique

```
<h1>Lister les libellés d'un document</h1>

<p>Nous allons apprendre à lister les libellés d'un document HTML à l'aide du DOM niveau 2 (noyau et HTML). On se reposera essentiellement sur <code>document.getElementsByTagName</code>. Cette méthode, je cite, <q cite="http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-A6C9094" lang="en">Returns a <code>NodeList</code> of all the <code>Element</code>s with a given tag name in the order in which they are encountered in a preorder traversal of the <code>Document</code> tree.</q>.</p>

<p>Qu'est-ce qu'une <code>NodeList</code>? C'est une interface d'énumération de nœuds. La spécification la décrit ainsi:</p>

<b><blockquote cite="http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-536297177" lang="en"></b>
    <p>The <code>NodeList</code> interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented. <code>NodeList</code> objects in the DOM are live.</p>
    <p>The items in the <code>NodeList</code> are accessible via an integral index, starting from 0.</p>
</blockquote>

<p>Voici le code nécessaire. Par simplicité, on suppose qu'une fonction <code>getInnerText</code> existe:</p>

<pre class="code">
<b>function</b> printLabelsInfo() {
    <b>var</b> msg = '';
    <b>var</b> labels = document.getElementsByTagName('label');
    <b>for</b> (<b>var</b> index = 0; index < labels.length; ++index) {
        <b>var</b> label = labels[index];
        msg += getInnerText(label);
        <b>if</b> (label.hasAttribute('for') && document.getElementById(label.htmlFor))
            msg += ' ->' + label.htmlFor;
        <b>if</b> (label.hasAttribute('accesskey'))
            msg += ' (Alt' + label.accessKey)';
        msg += '\n';
    }
    alert(msg);
} <i>// printLabelsInfo</i>
</pre>

<p>Sauvegardez ce code dans un fichier <tt>labels.js</tt>. Vous remarquez qu'il est de faible taille:</p>
<pre class="console">
```

```
<samp>$</samp> <kbd>ls -l</kbd>
<samp>total 4
-rw-r--r-- 1 demo demo 530 2006-08-23 16:58 labels.js
$</samp>
</pre>
```

- On voit ici des exemples de nombreuses balises sémantiques. On le voit, code représente du code de façon générale (lorsqu'il s'agit d'un nom de variable dans un code source, on utilise de préférence var), et tt sert à indiquer les textes perçus comme du contenu informatique mais qui ne sont pas forcément affichés par le système (samp), ni saisis (kbd). Classiquement, les noms de fichiers.
- Dans un code source, on peut vouloir améliorer la mise en page, par exemple en mettant les mots réservés en gras et les commentaires en italique, et peut-être en gris. Ce n'est que de la coloration syntaxique, sans sémantique : i et b sont parfaits pour cela !
- Remarquez les citations brèves, en ligne, avec q (et son attribut cite pour avoir la source), et celles plus longues, en bloc, avec blockquote (et son attribut cite, là encore).

Là aussi, le HTML et la CSS ne totalisent que 3 Ko. Voici le résultat, doté d'une légère feuille de styles.

Figure A-4

Notre fragment de documentation avec un style léger



Pour aller plus loin...

Livres

Réussir son site web avec XHTML et CSS, 2^e édition

Mathieu Nebra

Editions Eyrolles, mars 2008, 316 pages

ISBN 978-2-212-12307-4

HTML avec CSS et XHTML Tête la première

Elisabeth Freeman, Eric Freeman

O'Reilly France, août 2006, 720 pages (de bonheur)

ISBN 2-841-77413-9

Introduction à HTML et CSS

Eric Sarrion

O'Reilly, février 2006, 231 pages

ISBN 2-841-77400-7

HTML et CSS 2

Molly Holzschlag

Pearson Education, septembre 2005, 330 pages

ISBN 2-744-01994-1

Design web : utiliser les standards

Jeffrey Zeldman

Eyrolles, avril 2005, 414 pages (d'évangile)

ISBN 2-212-11548-2

HTML : précis et concis

Jennifer Niederst

O'Reilly, avril 2002, 122 pages (d'aide-mémoire)

ISBN 2-841-77157-1

Sites

- Le W3C évidemment : <http://w3.org>. Et les recommandations fondamentales (seule la version anglaise a valeur de référence) :
 - HTML 4.01 : <http://w3.org/TR/html401/> (version française à cette adresse : <http://www.la-grange.net/w3c/html4.01/cover.html>)
 - XHTML 1.0 : <http://w3.org/TR/xhtml1/> (version française à cette adresse : <http://www.la-grange.net/w3c/xhtml1/>)

- OpenWeb, un excellent site didactique en français, joli et pratique :
<http://openweb.eu.org/>. On notera en particulier :
 - http://openweb.eu.org/articles/xhtml_une_heure/
 - http://openweb.eu.org/articles/respecter_semantique/
 - http://openweb.eu.org/articles/html_au_xhtml/
 - http://openweb.eu.org/articles/formulaire_accessible/
- Pompage, excellent site voué à la traduction francophone d'articles de grande qualité sur la conception web : <http://pompage.net>. On consultera tout particulièrement, histoire de se mettre en appétit, les articles suivants :
 - <http://pompage.net/pompe/bon pied standards/>
 - <http://pompage.net/pompe/cederholm/>
 - <http://pompage.net/pompe/separation/>
 - <http://pompage.net/pompe/autableau/>
 - <http://pompage.net/pompe/listes/>
 - <http://pompage.net/pompe/listesdefinitions/>
 - <http://pompage.net/pompe/doctypecontenttype/>
- AccessiWeb, la cellule accessibilité de BrailleNet, regorge de documentations et manuels pratiques en français pour rendre vos sites et créations numériques plus accessibles : [http://www.accessiweb.org/](http://www.accessiweb.org).
- Opquast est un référentiel français de bonnes pratiques de conception web classées suivant de multiples axes, doté en plus d'un outil d'aide à l'estimation et au suivi de la qualité de vos sites : [http://www.opquast.com/](http://www.opquast.com).
- Alsa Créations est un site didactique plein d'excellents articles sur les standards, XHTML et CSS : [http://www.alsacreations.com/](http://www.alsacreations.com). Un forum permet par ailleurs aux visiteurs de s'entraider.
- A List Apart est un incontournable : [http://alistapart.com/](http://alistapart.com).

B

Aspect irréprochable et flexible : CSS 2.1

Cette annexe n'a pas l'intention de vous apprendre CSS. Le sujet est si vaste, et les subtilités si nombreuses, qu'il faudrait pour cela des livres entiers (et il en existe d'excellents, voir en fin d'annexe). Non, il s'agit juste de faire le point sur la prise en charge des CSS dans les navigateurs, de rappeler quelques grandes notions fondamentales (structure des règles, cascade, modèles fondateurs), et de faire un tour d'horizon des sélecteurs et des propriétés.

Notez que ce survol sera moins détaillé que pour l'annexe A, dont l'objectif était d'attirer votre attention sur les bons et les mauvais usages fréquents et de mettre en avant des attributs sous-employés, ce qui exigeait de longues descriptions et quelques exemples solides. Ici, on a plutôt une sorte de référence laconique.

Statut, état et vocabulaire

CSS est un standard W3C, publié sous forme de recommandations. À l'exception de (X)HTML, il n'y a sans doute pas de standard du Web plus utilisé que CSS.

Les versions de CSS

Tableau B-1 Les versions du standard CSS

Version	Description
1.0	Première édition le 17 décembre 1996 (eh oui, les CSS ont dix ans !), avec une révision le 11 janvier 1999. Bien qu'assez simple, cette première mouture fournissait déjà l'essentiel des modèles de boîte et de mise en forme visuelle.
2.0	Sortie le 12 mai 1998, cette version ajoutait énormément de choses : les types de média, la valeur spéciale <code>inheri t</code> , la pagination, des fonctionnalités d'internationalisation, une gestion détaillée des tableaux, les positionnements absolu, relatif et fixe, la gestion du débordement et de la troncation de contenu, le contenu généré, et j'en passe.
2.1	Toujours à l'état d'ébauche (cinquième révision le 11 avril 2006), cette version constitue surtout un affinage et une série de corrections et de précisions à la version 2.0. Elle constitue l'état « actuel » de CSS dans la plupart des navigateurs.
3.0	Dans la tendance actuelle de modularisation au W3C, CSS 3 est constituée de 37 (oui, vous avez bien lu) modules, chacun faisant l'objet d'un travail indépendant. À l'heure où j'écris ces lignes, aucun n'est totalement finalisé ! Huit le sont presque (CR, <i>Candidate Recommandation</i>), trois sont au stade précédent (<i>last call</i>), dont celui des sélecteurs (extrêmement attendu !) et celui des polices de caractères, et le reste est en ébauche (parfois figée depuis des années), voire même pas démarré (cinq modules). En somme, le travail n'avance pas.

Prise en charge actuelle

Comme on peut le voir dans l'avant-propos, la prise en charge de CSS est généralement bonne, avec toutefois quelques écarts. Pour simplifier, on peut dire qu'avec la sortie de MSIE 7, tous les navigateurs véritablement répandus prenaient décemment en charge CSS 2.1.

Voici tout de même un petit bilan (mais n'hésitez pas à consulter les informations techniques de vos navigateurs cibles : ces données évoluent souvent). Attention : les informations pour MSIE supposent une page en mode strict (utilisation d'un DOCTYPE sur une DTD (X)HTML Strict). Voir l'annexe A à ce sujet.

Les adjectifs expriment le degré de prise en charge.

Navigateur	CSS 1	CSS 2.1	CSS 3
MSIE 6	Bon	Moyen	-
MSIE 7	Très bon	Bon	-
MSIE 8 b1	Très bon	Très bon	Léger
Mozilla, Firefox, Camino	Très bon	Très bon	Partiel
Safari	Très bon	Très bon	Léger
Opera	Très bon	Très bon	Léger

Les parties les moins bien prises en charge de CSS 2.1 concernent surtout les feuilles de styles auditives (*aural style sheets*), qui visent à contrôler la lecture vocale des pages : sexe de la voix, richesse, prosodie, intonation, débit... Ce ne sont pas vraiment les navigateurs qui doivent implémenter cela, mais les lecteurs d'écran !

Le jargon CSS

Lorsqu'on parle de CSS, certains termes précis reviennent régulièrement. Reprenons-les ici avant de poursuivre la lecture de cette annexe.

- Une **propriété** est un nom spécifique représentant un aspect CSS. Par exemple, `font-weight` gère l'épaisseur des caractères (normale, grasse, etc.) tandis que `white-space` régit le traitement des espacements et retours chariot. Chaque propriété dispose d'une série de valeurs possibles, avec généralement plusieurs syntaxes autorisées. Elle peut avoir une valeur par défaut, bénéficier ou non de l'héritage par cascade, et ne s'appliquer qu'à certains éléments.
- Un **sélecteur** est un texte respectant une syntaxe spéciale qui permet de désigner des éléments de la page en fonction de critères variés : leur nom de balise, leur ID, leurs classes CSS, la présence d'un attribut, leur position dans le document, la langue de leur contenu, etc. En combinant des sélecteurs, on peut décrire une extraction très fine et très précise des éléments dans le document.
- Une **règle** est l'élément de base d'une feuille de styles CSS : elle définit une série de propriétés pour les éléments désignés par un ou plusieurs sélecteurs.

Prêt pour un exemple complexe ? C'est parti ! Dans le code CSS suivant :

```
#examples tr.critical>th:first-child, th.critical {
    background: red;
    color: white;
    font-weight: bold;
}
```

- `#examples` est un sélecteur d'ID, l'espace qui suit est un sélecteur de descendant, `tr` est un sélecteur de type, `.critical` est un sélecteur de classe, le chevron fermant (`>`) est un sélecteur d'élément fils, `th` est un sélecteur de type, `:first-child` est une pseudo-classe et la virgule est un opérateur de groupement entre deux combinaisons de sélecteurs. Si vous êtes perdu, ne vous en faites pas, on reverra ces sélecteurs plus loin.
- `background`, `color` et `font-weight` sont des propriétés, dont `red`, `white` et `bold` sont les valeurs respectives.
- L'ensemble du fragment constitue une règle.

Histoire de ne pas vous faire languir si vous n'êtes pas très au point côté CSS, la règle s'applique ici aux éléments suivants :

- Les éléments `th` apparaissant comme premier élément fils dans les `tr` ayant une classe CSS `critical`, lesquels apparaissent quelque part dans un élément d'ID `examples`.
- Les éléments `th` ayant une classe CSS `critical`.

Bien comprendre la « cascade » et l'héritage

Les feuilles de styles applicables à un document peuvent avoir trois origines :

- **Le navigateur**, qui fournit généralement une feuille de styles par défaut (grâce à laquelle les titres sont plus gros, en italique, etc.).
- **L'auteur** du document ; la feuille est alors généralement un fichier `.css` associé par une balise `link`.
- **L'internaute**, qui peut appliquer une feuille de styles utilisateur (pour peu que son navigateur l'y autorise).

La cascade détermine comment sélectionner, propriété par propriété, la valeur spécifique à retenir pour un élément donné. Contrairement à une idée répandue, elle ne décrit pas le mécanisme d'héritage, bien plus simple et qui n'a pas besoin d'elle. Je reviendrai sur cet héritage un peu plus tard.

Le sens de la cascade

La cascade se déroule approximativement de la façon suivante :

- 1 On récupère toutes les règles applicables (en vertu de leurs sélecteurs) à l'élément concerné, pour le média courant (saviez-vous qu'on peut restreindre une feuille de styles à certains médias seulement, comme l'écran, l'imprimante ou la vidéo-projection ?).

- 2** On les trie par importance (présence de la clause `!important`) et par origine, dans l'ordre suivant (du moins prioritaire au plus prioritaire) :
- règles « par défaut » venant du navigateur ;
 - règles normales de l'internaute ;
 - règles normales de l'auteur ;
 - règles importantes de l'auteur ;
 - règles importantes de l'internaute ;
- 3** À priorité égale, on les trie par spécificité (voir section suivante).
- 4** Enfin, on trie par ordre d'apparition : quand plusieurs définitions ont la même priorité et la même spécificité, la dernière est retenue.

La syntaxe `!important` en fin de déclaration de propriété est là pour améliorer l'accessibilité en garantissant à l'internaute la possibilité de remplacer une propriété qui nuit à son confort : il suffit de déclarer une autre valeur dans une feuille de styles utilisateur, dotée de l'attribut final `!important`.

Calcul de la spécificité d'une règle

Voyons à présent comment calculer la spécificité, ou le « poids », d'une règle CSS. La spécificité est une valeur à quatre composantes : `a-b-c-d`. Ces composantes étant souvent de valeur inférieure à 10, on a en général un nombre décimal entre 0 et 9999. C'est le *poids*. Mais comment déterminer la valeur de chaque composante ?

- La composante `a` vaut 1 pour un attribut HTML `style`, 0 pour une feuille externe.
- La composante `b` est le nombre de sélecteurs d'ID impliqués (`#truc`).
- La composante `c` est le nombre de sélecteurs sur attributs (`[truc]`, `.bidule`), ainsi que les pseudo-classes (par exemple `:hover` ou `:focus`).
- La composante `d` est le nombre de sélecteurs de type (par exemple `h1`) et de pseudo-éléments (par exemple `:first-line`).

Vous avez besoin d'exemples ? D'accord !

Tableau B-2 Exemples de calculs de spécificité

Sélecteurs	a	b	c	d	Total
*	0	0	0	0	0
li	0	0	0	li → 1	1
li:first-line	0	0	0	li, :first-line → 2	2
ul li	0	0	0	ul, li → 2	2
ul ol + li	0	0	0	ul, ol, li → 3	3

Tableau B-2 Exemples de calculs de spécificité (suite)

Sélecteurs	a	b	c	d	Total
h1 + *[rel=up]	0	0	[rel=up] → 1	h1 → 1	11
ul ol li.red	0	0	.red → 1	ul, ol, li → 3	13
li.red.level	0	0	.red, .level → 2	li → 1	21
#x34y	0	#x34y → 1	0	0	100
style="..."	1	0	0	0	1000

Que se passe-t-il avec la présentation dans HTML ?

Le détail obscur : si, horreur ultime, le HTML contenait des attributs de présentation (vous savez, ces momies que sont `bgcolor`, `border`, `align`, etc.), le navigateur peut les prendre en compte à condition de les considérer comme des règles placées en début de feuille de styles auteur, avec une spécificité zéro. En d'autres termes, elles seront prioritaires uniquement sur le style par défaut fourni par le navigateur et céderont le pas devant toute spécification de type CSS.

L'héritage

L'héritage est un mécanisme par lequel des éléments voient certaines de leurs propriétés « hériter » leur valeur de celle utilisée par un élément conteneur.

Par exemple, si `body` précise une `font-size` de valeur `large`, les éléments `p` à l'intérieur de `body` (directement ou non) utiliseront la même valeur pour leur propre `font-size`, sauf instruction contraire.

Pour bien comprendre le rôle de l'héritage, il faut examiner comment le navigateur détermine la valeur concrète d'une propriété. Il ne suffit pas de prendre la valeur spécifiée dans la CSS, loin de là !

Pour commencer, sachez qu'un navigateur doit avoir défini une valeur concrète pour toutes les propriétés de tous les éléments lorsqu'il a achevé le *rendering* de la page ! Cela fait donc beaucoup de valeurs, même si on exclut les propriétés qui ne s'appliquent pas au média courant (par exemple, les propriétés de pagination lorsqu'on affiche à l'écran).

Cette valeur effective (*actual value* dans la spécification) est le résultat d'un calcul en quatre étapes. Dit comme cela, ça semble très compliqué, mais vous allez voir, c'est en fait plutôt naturel.

On commence par déterminer la valeur spécifiée, celle qui fait l'objet d'une déclaration ou d'une valeur par défaut. Pour cela, on cherche une valeur indiquée explicitement dans les CSS, en résolvant les conflits éventuels à l'aide de l'algorithme de cascade décrit ci-dessus.

C'est là que l'héritage peut entrer en jeu. Chaque propriété peut ou non en bénéficier automatiquement : cela est précisé dans la spécification CSS 2.1, à l'aide de la caractéristique *Inheritable*. Si on n'a pas trouvé de valeur explicite pour une propriété « héritable », on utilise alors la valeur calculée de l'élément parent (pour lequel toutes les propriétés ont déjà été résolues). On peut obtenir le même effet pour une propriété sans héritage automatique, en définissant la propriété avec la valeur spéciale `inherit`.

De la valeur spécifiée à la valeur concrète

En revanche, s'il n'y a pas d'héritage (la propriété n'est pas définie et elle ne bénéficie pas d'un héritage automatique), on utilise alors la valeur initiale, indiquée dans la spécification CSS 2.1 pour cette propriété.

Deuxième étape : obtenir la **valeur calculée**. Il s'agit d'ajustements à la valeur spécifiée, par exemple la transformation d'URI relatifs en URI absolu, la conversion de tailles exprimées en unités relatives (`em`, `ex`...) en unités absolues (`px`), et tout autre changement qui n'a pas besoin d'un véritable *rendering* pour être déterminé. Cette valeur existe pour tout élément, même si la propriété ne s'applique pas à l'élément lui-même, afin de pouvoir utiliser l'héritage sur ses éléments descendants.

Troisième étape : passer à la **valeur utilisée**. On effectue alors les conversions résultant d'un *rendering*, comme transformer une largeur exprimée en pourcentage en une largeur absolue (puisque l'on connaît alors la largeur réelle de son conteneur).

Ultime étape, qui n'intéresse d'ailleurs pas toujours le développeur web : la **valeur effective (ou concrète)**, celle qui sera réellement employée. C'est le résultat des contraintes externes (notamment celles du périphérique) sur la valeur utilisée. Par exemple, une largeur absolue peut encore être de 2,8 pixels, alors qu'à l'écran, on ne peut afficher que des pixels entiers ; on arrondira donc à 3. Une police pourrait demander du 13 points, mais si le système de gestion des polices ne peut obtenir cette taille exacte, on ajustera sur la taille la plus proche, par exemple 12 points. Le document est en couleurs, mais l'imprimante n'autorise que les nuances de gris ; on interpolera donc les valeurs vers de telles nuances. Vous voyez l'idée.

Voilà certainement plus d'informations que vous ne souhaitez en avoir. Il est vrai que ce sont surtout les deux premières étapes qui nous intéressent !

Les modèles de boîte et de mise en forme visuelle

Je vous l'annonce sans ambages : nous n'irons pas dans tous les détails, loin s'en faut. Le modèle de boîte (*box model*) a fait couler tant d'encre et d'octets qu'on pourrait remplir un volume encyclopédique avec la somme des articles, chapitres et billets sur le sujet.

Je vais simplement couvrir la partie émergée, l'essentiel, le courant, sans me soucier des cas particuliers.

Les côtés d'une boîte : ordre et syntaxes courtes

De très nombreuses propriétés CSS peuvent s'appliquer sur les quatre côtés d'une boîte, ou sur ses côtés horizontaux (haut et bas) et verticaux (droite et gauche), ou sur ses quatre côtés individuellement. Vous trouverez alors toujours des propriétés dites « abrégées » (*shorthand properties*), permettant de spécifier tous ces côtés d'un seul coup.

Les variantes et l'ordre sont toujours les mêmes, aussi les précisé-je ici une bonne fois pour toutes. Prenons l'exemple de la propriété `margin`, qui régit la marge externe d'une boîte, comme nous le verrons à la prochaine section.

Il existe des propriétés dédiées pour chaque côté : `margin-top`, `margin-right`, `margin-bottom` et `margin-left`. Il existe aussi la propriété courte `margin`, justement, qui peut prendre les trois formes suivantes.

Tableau B-3 Variantes de définition d'une propriété courte

Variante de format	Résultat
<code>margin: 1em;</code>	Même marge pour les quatre côtés : 1em.
<code>margin: 1em auto;</code>	Marge de 1em en haut et en bas, marge automatique (centrage de bloc) à droite et à gauche.
<code>margin: 1em 0 0.5em</code>	Marge haute de 1em, droite et gauche de 0, basse de 0.5em
<code>margin: 0.5em 0 1em 2ex;</code>	Marge haute de 0.5em, droite de 0, basse de 1em, gauche de 2ex.

Si vous avez du mal à retenir l'ordre, pensez à ceci : on part du haut et on suit les aiguilles d'une montre.

Unités absolues et relatives

CSS fournit de nombreuses unités pour exprimer les tailles (de police, de bloc...). Certaines sont absolues et d'autres relatives. Certaines n'ont de sens que pour certains médias. En voici un récapitulatif.

Tableau B-4 Unités de CSS 2.1

Unité	Nature	Signification
%	relative	Pourcentage de la valeur calculée héritée (ou initiale, pour body). Par exemple dans un body avec <code>font-size: 1.5em</code> , un p avec <code>font-size: 150 %</code> aura en fait un <code>font-size</code> de 2.25em.

Tableau B-4 Unités de CSS 2.1 (suite)

Unité	Nature	Signification
cm	absolue	Centimètres. S'ajuste normalement aux résolutions de l'imprimante comme de l'écran (72 ou 96 ppp).
em	relative	La hauteur de la police courante, définie comme le <i>em square</i> typographique. En gros, en dépit de ses origines liées à la largeur, c'est à peu près la hauteur d'un caractère majuscule. Unité de choix pour les marges et espacements en général (même si j'ai plus tendance à limiter aux côtés haut et bas) et aux hauteurs de blocs.
ex	relative	La hauteur d'un caractère minuscule sans jambe (on utilise généralement « x »). Correspond souvent à la largeur moyenne d'un caractère : un conteneur de largeur 10ex peut contenir environ 10 caractères. Unité pertinente aussi pour les marges et espacements en général (je l'utilise surtout pour les bords droit et gauche, et les largeurs de blocs).
in	absolue	Pouces (<i>inches</i>). Je rappelle qu'un pouce vaut 2,54 cm. Même remarque que pour cm.
mm	absolue	Millimètres. Même remarque que pour cm.
pc	absolue imprimante	Picas (unité typographique). Cette merveilleuse unité semi-préhistorique vaut 12 points (voir pt).
pt	absolue imprimante	Points. Unité typographique qui n'a vraiment de sens que pour une CSS d'impression (média print). CSS 2.1cale le point à 1/72 de pouce, soit environ 0,35 mm. Pas étonnant que beaucoup préfèrent le système métrique aux unités britanniques. Notez que lors de l'impression, caler les tailles de fontes en points est bien plus pertinent qu'en em/ex, car cela favorise l'homogénéité sur un grand nombre d'imprimantes et de systèmes.
px	classée relative, mais plutôt « absolue écran »	Pixels. Unité un peu fourbe, car selon la résolution de l'écran (pas 1024×768 , mais 72 ppp), un pixel sera plus ou moins gros. Sur Mac OS (avant OS X) notamment, les pixels étaient notoirement plus petits, ce qui rendait toute police 10px illisible. N'utilisez cette unité que pour des éléments dont la taille doit coller à des ressources non redimensionnables, comme des images. Pour du texte, je suis partisan de la proscrire, surtout sur des petites et moyennes tailles (moins de 30px), car le texte ne peut alors pas être redimensionné (notamment agrandi) suivant les besoins de l'internaute.

Je vous recommande très chaudement d'utiliser des **unités relatives** pour tout ce qui a vocation à suivre la taille du texte. Cela inclut souvent les bordures, les marges, les positionnements, et suivant l'esthétique retenue, parfois les espacements (*paddings*).

Ainsi, l'ensemble de votre page s'ajustera mieux lorsque l'internaute modifiera la taille de base des polices de caractères pour y voir plus clair, ce que tous les navigateurs répandus permettent de faire.

On donnera, notamment, priorité à em, ex et % à l'écran, et pt à l'impression !

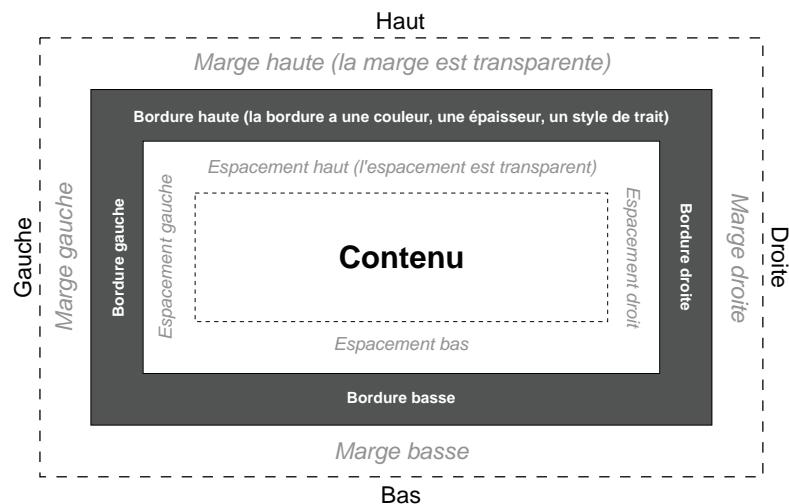
Dernière remarque : pour une taille de zéro, on ne précise pas l'unité en général. C'est en effet mal vu par les esthètes (on parle de *bad form*), car zéro est zéro, quelle que soit l'unité.

Marge, bordure et espacement

Une boîte est dotée, de l'extérieur vers l'intérieur, d'une marge (espacement externe à sa bordure), d'une bordure (avec une couleur, un type de trait et une épaisseur), d'un espacement (*padding* en anglais, espacement interne à la bordure), et tout au centre, de son contenu. Le schéma suivant illustre cette imbrication :

Figure B-1

Imbrication de marge, bordure, espacement et contenu



À présent, un point très important, car contre-intuitif : la largeur et la hauteur d'un élément sont en réalité les dimensions de son contenu. Prenons la règle suivante :

```
div#demo {
    width: 40ex;
    padding: 1ex;
    border: 1ex solid gray;
    margin: 1em 1ex;
}
```

Le div d'ID `demo` aura une « largeur visuelle » de 44ex ! En effet :

$$40\text{ex} + 2 \times 1\text{ex} + 2 \times 1\text{ex} = 44\text{ex}$$

Il s'agit de sa largeur de contenu, à laquelle on ajoute les largeurs des espacements gauche et droit ainsi que celles des bordures gauche et droite.

Éléments en ligne et de type bloc

La plupart des éléments censés faire partie du flux d'un texte sont dits « en ligne » (*inline*). Ceux censés apparaître comme des blocs hors du flux de texte sont dits « de type bloc » (*block-level*).

Il existe en réalité toute une gamme de cas spécialisés, définis par la propriété `display`. CSS 2 a notamment ajouté une série de valeurs spécifiques aux composants de tableaux de données.

Des éléments naturellement en ligne (par exemple `span`, `em`, `strong`, `code`) peuvent devenir de type bloc à l'aide de la propriété `display`, et réciproquement.

Je mentionne cela parce qu'un élément en ligne a certaines limitations en terme de modèle de boîte. Par exemple, il n'a pas de marge verticale (haute ou basse).

Éléments remplacés

Voici l'astuce du jour... Si vous lisez la spécification CSS 2.1, vous trouverez un peu partout le terme « élément en ligne non remplacé » (*non-replaced inline element*). On peut légitimement se demander à quoi rime ce « non remplacé », qu'on voit un peu partout. La réponse se cache à la section 3.1 de la recommandation, dont le titre, « Conformité », ne nous l'aurait pas laissé supposer.

Un élément remplacé est un élément hors du champ d'action du formateur CSS, en tout cas en ce qui concerne ses dimensions propres. Dans la pratique, c'est l'image obtenue par chargement d'une balise `img` ou alors le contenu chargé par une balise `object`. Tout autre élément en ligne est donc non remplacé, ce qui rend cette précision superflue dans la plupart des cas.

Fusion des marges

Les marges expriment généralement un besoin d'espace autour d'un élément. Cependant, dans le cas des marges verticales, il est souvent inutile de cumuler les marges. Imaginons qu'on utilise la règle suivante :

```
p { margin: 1em 0; }
```

Cette règle demande des marges haute et basse, dites « marges verticales », de `1em` (hauteur moyenne d'un texte dans la police de caractères active), et pas de marges horizontales (droites et gauches).

Pourtant, lorsqu'on va enchaîner deux paragraphes, il serait curieux, voire agaçant, que ces marges se cumulent, produisant un espacement de 2 lignes de haut entre les paragraphes. Ce n'est généralement pas l'intention de l'auteur de la règle. Pour cette raison, dans de nombreuses situations, CSS 2.1 fusionne les marges verticales. Pour faire simple, lorsqu'on a deux marges verticales adjacentes, on n'utilise que la plus grande des deux.

Attention toutefois, ce mécanisme n'a pas cours dans tous les cas. La section 8.3.1 de la recommandation W3C détaille les cas concernés, dont la liste exhaustive va au-delà du cadre de cette annexe.

Le modèle W3C et le modèle Microsoft

Voici la source de bien des soucis, en tout cas jusqu'à récemment (de nos jours, à peu près tous les professionnels ont pris le pli). Les développeurs de MSIE avaient à l'origine mal interprété le sens des propriétés `width` et `height`. Ils pensaient que ces dimensions incluaient l'espacement et la bordure.

Ce n'est pas déraisonnable : dans la pratique, quand vous faites référence aux dimensions d'une boîte, vous parlez des bords extérieurs de la boîte, pas de la taille du contenu calé dans la boîte par des protections en mousse ou en polystyrène. Hélas, ce n'est pas le sens retenu par la recommandation W3C.

Le souci est apparu dès que les autres navigateurs ont rattrapé MSIE 5 sur le terrain des CSS. Ces navigateurs implémentaient correctement la recommandation, et on s'est retrouvé avec des incohérences flagrantes entre l'affichage MSIE et celui des autres navigateurs. Je m'en souviens encore, et croyez-moi, ça nous a tous fait fulminer.

MSIE 6 a donc sorti une astuce, le *doctype switching*, expliquée au début de l'annexe A, astuce qui a rapidement été reprise par les autres navigateurs pour laisser l'auteur de la page décider de l'interprétation de `width` et `height`.

Lorsqu'une page déclare une DTD stricte (HTML 4.01 Strict ou XHTML 1.0 Strict, par exemple), le navigateur fonctionne en *Strict Mode*. On respecte alors la définition du W3C. Si une page ne déclare pas de DTD (rhôôô...) ou déclare une DTD non stricte, le navigateur utilise le *Quirks Mode*, qui simule le comportement de l'ancien MSIE en considérant que l'espacement et la bordure sont compris dans la taille.

En réalité, de nombreux navigateurs utilisent aussi le *Quirks Mode* pour émuler d'autres erreurs CSS de MSIE, et pas seulement de la version 5.5... Mais c'est un autre débat.

En conclusion : déclarez toujours une DTD, et qu'elle soit stricte, bon sang !

Tour d'horizon des sélecteurs

CSS 2.1 définit de nombreux sélecteurs (et si vous voyiez CSS 3 !). En voici un tour d'horizon. Attention, certains ne sont pas pris en charge par MSIE 6, voire également ignorés par quelques autres navigateurs.

Tableau B–5 Les sélecteurs de CSS 2.1

Sélecteur	Description
*	Sélecteur universel. Correspond à tous les éléments.
balise	Sélecteur de type. Correspond aux éléments ayant ce nom de balise.
balise1 balise2	Sélecteur de descendants. Correspond aux éléments <code>balise2</code> situés quelque part à l'intérieur d'un élément <code>balise1</code> , à quelque niveau de profondeur que ce soit.
.classe	Sélecteur de classe. En (X)HTML, équivalent à <code>*[class~="classe"]</code> (voir plus bas).
#unID	Sélecteur d'ID. Correspond à l'élément dont l'attribut <code>id</code> possède la valeur <code>unID</code> .
balise:link balise:visited	Pseudoclasses de lien. Correspond aux éléments <code>balise</code> (qui doivent être des sources de lien, donc généralement des éléments a) dont la cible n'a pas encore (ou a déjà, dans le second cas) été visitée.
balise:hover balise:active balise:focus	Pseudoclasses dynamiques. Correspond aux éléments <code>balise</code> à certains moments de l'interaction utilisateur. Respectivement, il s'agit d'un survol souris, d'une activation (clic ou touche Entrée, par exemple) ou de l'obtention du focus (arrivée sur l'élément par tabulation, par exemple).
balise1 > balise2	Sélecteur de fils. Correspond aux éléments <code>balise2</code> dont l'élément père est un élément <code>balise1</code> . Plus restrictif donc que l'espace, qui n'a pas de limite de profondeur.
balise1 + balise2	Sélecteur d'élément adjacent. Correspond à un élément <code>balise2</code> dont le précédent élément frère est de type <code>balise1</code> . Par exemple, <code>h2 + p</code> sélectionne les paragraphes qui suivent immédiatement des titres de deuxième niveau.
balise:first-child	Pseudoclasse de premier fils. Correspond à tout élément <code>balise</code> étant le premier élément fils de son élément père. Par exemple, permet de sélectionner le premier paragraphe d'une série.
balise[attr]	Sélecteur d'attribut. Correspond à tout élément <code>balise</code> ayant un attribut <code>attr</code> (quelle qu'en soit la valeur, même vide).
balise[attr="value"]	Sélecteur d'attribut. Correspond à tout élément <code>balise</code> dont l'attribut <code>attr</code> a exactement la valeur <code>value</code> (sensible à la casse). Les guillemets sont obligatoires.
balise[attr~="part"]	Sélecteur d'attribut. Correspond à tout élément <code>balise</code> dont l'attribut <code>attr</code> contient une série de valeurs séparées par des espaces, et dont l'une est <code>part</code> . Voir l'équivalence du sélecteur de classe.
balise[attr = "prefix"]	Sélecteur d'attribut. Correspond à tout élément <code>balise</code> dont l'attribut <code>attr</code> contient une série de valeurs séparées par des tirets (-), et dont la première est <code>prefix</code> . Utile pour l'attribut <code>lang</code> , par exemple.
:lang(code)	Pseudoclasse de langue. Sélectionne tout élément dont la langue correspond à <code>code</code> .

Groupement

Afin de limiter la duplication (cauchemar de la maintenance), on peut affecter une série de propriétés à plusieurs sélections (combinaisons de sélecteurs) disjointes, en les séparant simplement par des virgules.

```
h1, h2, h3, h4, h5, h6 {
    font-weight: normal;
}
```

Dans cet exemple, tous les titres utiliseront une épaisseur normale au lieu de la graisse par défaut.

Pseudo-éléments

Un pseudo-élément correspond à une portion de texte à l'intérieur d'un élément, ou à un contenu généré dynamiquement. En tout état de cause, il s'agit d'un fragment du document qui ne correspond pas à un élément, mais fait néanmoins l'objet d'une sélection.

Tableau B-6 Les pseudo-éléments de CSS 2.1

Pseudo-élément	Description
E:first-line	Première ligne d'un corps de texte. E doit être élément de type bloc, <code>inline-block</code> , <code>table-caption</code> ou <code>table-cell</code> . On évitera donc <code>span</code> , par exemple.
E:first-letter	Première lettre d'un contenu textuel, sauf si cette lettre est précédée d'un autre contenu (image, tableau en ligne...). Toutes les propriétés CSS ne s'appliquent pas (voir section 5.12.2 de la recommandation). Principalement utilisé pour faire des lettrines. S'applique aux mêmes types d'élément que :first-line, plus au type <code>list-item</code> .
E:before E:after	Permettent de générer un contenu avant ou après le texte normal de l'élément, à l'aide de la propriété CSS <code>content</code> . Très utilisés pour assortir les liens de petits glyphes ou d'informations sur la langue (attribut HTML <code>hreflang</code>)... Très pratique aussi pour une feuille de styles dédiée à l'impression (média <code>print</code>), qui peut ainsi afficher, à côté du texte des liens, les URL ciblées.

Tour d'horizon des propriétés

Il est temps de faire le tour des propriétés. Pour toutes celles qui ont des versions courtes, j'ai adopté une notation à lignes multiples, avec les possibilités entre crochets, listées dans l'ordre de leur apparition au sein de la propriété courte.

Chaque ligne supplémentaire constitue une composante optionnelle. La propriété `border` est la plus riche, puisqu'elle va d'une seule propriété consolidée (`border`) jusqu'aux propriétés très ciblées (par exemple `border-left-style`).

Je précise aussi une bonne fois pour toutes que toutes les propriétés peuvent avoir la fameuse valeur `inherit`. Je ne le préciserais pas dans les tableaux.

De l'art de réaliser des CSS légères

CSS fournit de nombreuses syntaxes courtes pour simplifier et alléger vos feuilles de style. Nous avons déjà évoqué ce mécanisme dans le cadre des propriétés de marge, de bordure et d'espacement. Allez donc consulter la recommandation pour le détail des autres propriétés courtes, notamment `font`, une vraie merveille !

Par ailleurs, toute propriété de couleur peut avoir les syntaxes suivantes :

- `transparent`, mot réservé, utile entre autres pour préciser une couleur de fond chaque fois qu'on précise une couleur de texte (évitant ainsi les avertissements des validateurs).
- `#rrvvbb`, où l'on représente les composantes rouge, verte et bleue par une valeur hexadécimale comprise entre `00` et `ff`, c'est-à-dire allant de `0` à `255`.
- `#rvb`, lorsque les composantes sont des doublons (par exemple `11`, `44` ou `ff`). Ces couleurs sont censées être plus fiables en rendu que les variantes détaillées (`#463`, équivalent de `#446633`, est censée être plus « garantie » que `#426431`, qui lui ressemble à s'y méprendre).
- `rgb(red, green, blue)`, où les composantes sont indiquées en base décimale, de `0` à `255`. À mon sens la plus verbeuse, donc à éviter.
- Mot réservé de couleur, par exemple `white` ou `red`. La recommandation définit en section 4.3.6 les 17 noms autorisés et leurs valeurs exactes.

N'oubliez pas qu'une feuille CSS plus courte est aussi plus rapide à charger, et pas forcément plus difficile à lire !

Propriétés du modèle de boîte : marges, espacements et bordures

Tableau B-7 Propriétés du modèle de boîte en CSS 2.1

Propriété	Description
border -[top right bottom left] -[width style color]	Bordures. L'épaisseur est exprimée en tant que taille ou à l'aide des mots réservés <code>thin</code> , <code>medium</code> et <code>thick</code> . On compte pas moins de 10 styles, les plus courants étant <code>solid</code> , <code>dotted</code> , <code>dashed</code> et <code>none</code> (qui diffère de <code>hidden</code> pour les cellules de tableaux !). Au fait, MSIE 7 cesse de dessiner les bordures <code>dotted</code> d'épaisseur 1px en <code>dashed</code> . Enfin !
margin -[top right bottom left]	Marges, donc espacements extérieurs.
padding -[top right bottom left]	Espacements intérieurs.

Propriétés de formatage visuel : positionnement, largeur, hauteur, baseline

Tableau B-8 Propriétés de formatage visuel en CSS 2.1

Propriété	Description
clear	Contrôle le flux autour d'un élément flottant. Peut valoir <code>none</code> , <code>left</code> , <code>right</code> ou <code>both</code> .
clip	Permet de restreindre la portion affichée d'un élément. Par défaut <code>auto</code> , donc affiche toute la boîte. Sinon, peut définir un rectangle avec <code>rect(top, right, bottom, left)</code> .
direction	Indique le sens du texte dans l'élément. Vaut <code>ltr</code> (de gauche à droite, par défaut) ou <code>rtl</code> .
display	Indique le type de boîte de l'élément. Les valeurs les plus courantes sont <code>inline</code> , <code>block</code> et <code>none</code> , mais il y en a 13 autres, dont 10 relatives aux tableaux !
float	Rend un élément flottant, c'est-à-dire l'extrait du flux normal du texte pour aller se caler quelque part contre les bords du conteneur, le flux du texte s'enroulant autour de lui. Suivant le côté du flottement, vaut <code>left</code> , <code>right</code> ou bien sûr <code>none</code> , sa valeur par défaut.
height, min-height, max-height	Régissent la hauteur d'un élément, en collaboration avec <code>overflow</code> .

Tableau B-8 Propriétés de formatage visuel en CSS 2.1 (suite)

Propriété	Description
line-height	Hauteur minimale de ligne, ce qui inclut le texte et un certain espacement au-dessus et en dessous du texte. Vaut normal par défaut (basé sur les caractéristiques de la police de caractères), sinon un nombre ou un pourcentage (multiplicateur de font-size), ou encore une taille spécifique (avec des unités, par opposition à un simple nombre).
overflow	Gère le dépassement de contenu vis-à-vis des dimensions souhaitées de la boîte. Vaut par défaut visible : la boîte s'adapte, notamment en hauteur. Peut aussi valoir hidden (le contenu est tronqué), scroll (barres de défilement présentes qu'on en ait besoin ou non) ou auto (barres de défilement, si besoin).
position	Définit le positionnement de l'élément. Vaut static par défaut (positionnement défini par le navigateur suivant les contraintes actives), mais peut aussi valoir absolute, relative ou fixed (lequel est enfin pris en charge par MSIE 7).
top, right, bottom, left	Pour un élément positionné, fournit les positions de ses extérieurs de marge (absolues ou relatives, suivant position).
unicode-bidi	Permet à la valeur de direction de fonctionner également sur un élément en ligne.
vertical-align	Fournit l'alignement vertical du contenu en ligne (ou du contenu d'une cellule de tableau). Les valeurs ont toutes rapport aux métriques de typographie : le défaut est baseline, et on en a 7 autres dont middle et text-top, plus la possibilité d'une taille ou d'un pourcentage.
visibility	Affiche ou masque un élément, tout en conservant l'espace occupé (contrairement à display). Outre les valeurs visible et hidden, une valeur collapse a un sens spécial pour les lignes et groupes (de lignes ou de colonnes) des tableaux.
width, min-width, max-width	Régissent la largeur d'un élément.
z-index	Utilisée uniquement sur les éléments positionnés. Indique leur position « verticale », entre l'œil de l'internaute et le document si vous préférez. Peut valoir auto (défaut) ou un numéro. Règle les questions de recouvrement entre éléments se superposant, par exemple lors d'un glisser-déplacer.

Propriétés de contenu généré automatiquement

CSS 2 a introduit la notion de contenu automatique, principalement sur trois axes :

- Du contenu entièrement synthétisé par la CSS, généralement présent devant ou derrière le contenu natif de l'élément.
- Des indices incrémentaux ; principalement pour les listes, mais aussi pour les titres par exemple (enfin des titres numérotés automatiquement, et hiérarchiquement si on le veut !).
- Des symboles ou images destinés aux listes à puces.

Tableau B-9 Propriétés de contenu automatique en CSS 2.1

Propriété	Description
content	Remplace le contenu de l'élément. On l'utilise principalement sur des sélecteurs de pseudo-éléments :before ou :after, pour ajouter plutôt que remplacer. Peut valoir une foule de choses : un texte fixe, une ressource externe dont on fournit l'URI, l'état d'un compteur, la valeur d'un attribut de l'élément (comme hreflang), l'ouverture ou la fermeture des guillemets ou encore le contrôle du niveau d'imbrication de ceux-ci.
counter-increment	Incrémente un ou plusieurs compteurs déjà définis. Utilise des paires nom × incrément, l'incrément étant optionnel et valant par défaut 1 (un). Voir l'exemple très parlant de la section 12.4 de la recommandation.
counter-reset	Définit ou réinitialise un ou plusieurs compteurs, en précisant éventuellement leurs nouvelles valeurs. Même syntaxe que counter-increment.
quotes	Définit les paires de guillemets à utiliser dans l'élément, niveau par niveau (pour des utilisations imbriquées). La valeur par défaut dépend du navigateur. none supprime tout guillemet (pas très utile...). Sinon, on précise des paires de textes, par exemple '«\00a0' '\00a0»' '''' pour le français (la séquence \00a0 représente une espace insérable). À utiliser en combinaison avec les valeurs open-quote et close-quote de content.
list-style -[type position image]	Régit l'apparence d'une liste. Le type peut avoir pas moins de 15 valeurs dont 11 de numérotation (listes ordonnées), 3 de puces, et none pour retirer les puces ou numéros. La position vaut inside ou outside (défaut), indiquant si les puces ou numéros s'affichent à l'intérieur ou à l'extérieur de la boîte des contenus. Enfin, image permet de remplacer les puces classiques par une image quelconque, dont on fournit l'URI.

Propriétés de pagination

Ces propriétés ne s'appliquent que dans le cadre d'un *rendering* sur média paginé, ce qui revient à dire : à l'impression (média print). On dispose alors d'une règle spéciale nommée @page, qui désigne la page physique et non pas un élément du document.

Tableau B-10 Propriétés de pagination en CSS 2.1

Propriété / pseudoclasse	Description
:first, :left, :right	Pseudoclasses utilisables sur @page pour régler par exemple les marges indépendamment pour la première page, les pages gauches (paire pour un document se lisant de gauche à droite) et les pages droites (impaires).
margin -[top right bottom left]	Marges classiques. Je les remets ici car elles ont un sens particulier lorsqu'on les applique à @page : ce sont alors les marges d'impression.

Tableau B-10 Propriétés de pagination en CSS 2.1 (suite)

Propriété / pseudoclasse	Description
orphans	Nombre minimum de lignes d'un bloc qui doivent apparaître en bas de page (lignes orphelines). Si le bas de la page est trop plein pour cela, le bloc démarre à la page suivante. La valeur est numérique, et vaut par défaut 2.
page-break-after page-break-before page-break-inside	Régissent les sauts de page. Appliquées à un élément, elles déterminent ce que le navigateur a le droit de faire après, avant et à l'intérieur de l'élément, respectivement. Les deux premières peuvent valoir auto (défaut), always (force le saut), avoid (éviter à tout prix), left ou right (forcer le saut jusqu'à une page gauche ou droite, par exemple pour un début de chapitre). La dernière ne peut valoir que auto ou avoid.
widows	Nombre minimum de lignes d'un bloc qui doivent apparaître en haut de page (lignes veuves). Si le bloc qui devait démarrer à la page précédente n'avait pas assez de lignes restantes pour ce haut de page, il démarre sur cette nouvelle page. La valeur est numérique, et vaut par défaut 2.

Propriétés de couleurs et d'arrière-plan

Tableau B-11 Propriétés de couleurs et d'arrière-plan en CSS 2.1

Propriété	Description
background -[color image repeat] attachment position]	Définit l'arrière-plan d'un élément. On a d'abord sa couleur, puis une image à utiliser, son mode de répétition (« mosaïque » : repeat par défaut, mais connaissez-vous repeat-x, repeat-y et no-repeat ?), son mode de défilement (scroll par défaut, mais connaissez-vous fixed ?) et sa position initiale dans l'élément (par exemple top right, ou 15% bottom, ou 2cm top).
color	Couleur du texte.

Propriétés de gestion de la police de caractères

Tableau B-12 Propriétés de la police de caractères en CSS 2.1

Propriété	Description
font -[style variant weight size family]	Régit la police de caractères. Voilà un cas où bien apprendre la syntaxe consolidée de la propriété courte (font) est payant ! Le style est généralement normal ou italic, la variante normal ou small-caps, le poids normal ou bold (presque aucun système de polices ne gère plus de 2 degrés de graisse...), la taille a une syntaxe plus complexe (voir ci-après) et la famille aussi. La propriété courte peut aussi utiliser juste un nom réservé de police système.

Taille de police

La taille peut être exprimée avec un mot-clé absolu, un mot-clé relatif, une taille classique ou un pourcentage de la taille de référence.

- Absolus : `xx-small`, `x-small`, `small`, `medium` (défaut), `large`, `x-large`, `xx-large`. Le rapport entre les valeurs successives n'est pas fixe, en particulier aux extrêmes.
- Relatifs : `smaller`, `larger`. Décale la taille sur l'échelle des absolus, et si on est déjà sur un extrême, interpole au mieux.

Dans le cas où la taille est précisée au sein de la propriété courte `font`, on dispose d'une syntaxe spéciale qui permet de faire d'une pierre deux coups en précisant à la volée le `line-height` : on utilise `font-size/line-height`, c'est très pratique et cohérent. Voir l'exemple un peu plus loin.

Famille de polices

La famille de polices permet de définir une série de polices à tenter d'utiliser, par ordre décroissant de préférence. Il s'agit de noms de polices que le navigateur va chercher sur le système de l'internaute. Les noms à espaces doivent être entre guillemets. Il est fortement conseillé, pour des raisons d'accessibilité, de terminer la série par un des noms génériques :

- `serif` : police à empattements, par exemple Times ;
- `sans-serif` : police sans empattements, par exemple Arial ;
- `cursive` : police à pleins et déliés, par exemple Monotype Corsiva ou Zapf Chancery ;
- `fantasy` : police « délirante », décalée, amusante ;
- `monospace` : police à chasse fixe, par exemple Courier.

Voici un exemple :

```
font-family: "Bitstream Vera Sans Mono", Monaco, monospace;
```

Tout spécifier d'un coup !

Enfin, voici un premier exemple de propriété courte, qui résume tout ce qu'on a besoin de dire sur la police :

```
font: 115%/1.4em "Bitstream Vera Sans Mono", Monaco, monospace
```

Ici, on ne précise ni le style, ni la variante, ni le poids, mais directement la taille (115 %), la hauteur de ligne (1.4em) et la famille.

Attention : certaines configurations partielles de style, variante et poids peuvent faire apparaître une ambiguïté : ainsi, si vous n'en précisez qu'une ou deux et utilisez la valeur `normal` pour la dernière, comment savoir de quelle propriété on parle ? Il faut alors être explicite, quitte à utiliser `inherit` pour maintenir les valeurs des propriétés qu'on ne souhaite pas affecter.

```
| font: normal 1.5em/1.8em sans-serif;
```

C'est ambigu : est-ce le type, la variante ou le poids qui est normal ?

```
| font: italic normal inherit/120%;
```

Et là, est-ce la variante ou le poids ?

```
| font: italic inherit normal inherit/120%;
```

Ici, pas de doute : c'est le poids, et on ne touche pas à la variante.

Dernier point : les noms réservés de polices système, qui configurent toute la police d'un coup. Cela permet de réaliser une interface cohérente avec celle du système d'exploitation. Les valeurs possibles sont :

- `caption`, utilisée pour les contrôles (composants visuels) à libellés (par exemple les boutons, les listes déroulantes).
- `icon`, utilisée pour labéliser les icônes (par exemple sur le bureau).
- `menu`, utilisée dans les menus (barres ou menus contextuels).
- `message-box`, utilisée pour les boîtes de dialogue à message.
- `small-caption`, utilisée pour labéliser les petits contrôles (e.g. comme les boutons de barre d'outils).
- `status-bar`, utilisée par les barres d'état.

Il suffit donc d'utiliser par exemple :

```
| div#status { font: status-bar; }
```

pour avoir un élément avec la fonte exacte des barres d'état sur le système de l'internaute.

Propriétés de gestion du corps du texte

Tableau B-13 Propriétés du corps du texte en CSS 2.1

Propriété	Description
letter-spacing	Espacement entre les lettres. Vaut zéro par défaut. Je conseille vivement de n'utiliser que des tailles en unité ex, très adaptée. Rien qu'à 0.1ex, on voit l'effet.
text-align	Alignment du texte dans un bloc. Peut valoir left, center, right ou justify.
text-decoration	Régit l'apparence de traits au-dessus ou en dessous du texte. Peut valoir none (par défaut), underline (soulignement), overline (trait au-dessus du texte), line-through (texte barré) ou... oserai-je le dire ? Bon, blink, mais gare au premier qui s'en sert ! C'est moche et tout le contraire d'accessible !
text-indent	Indentation de la première ligne d'un bloc de texte.
text-transform	Gère la casse. Peut valoir none (défaut), capitalize (initiales en majuscules, autres lettres inchangées), uppercase (majuscules) ou lowercase (minuscules). Voir aussi font-variant dans le tableau B-12.
white-space	Gestion des espacements dans le corps du texte. Voir ci-après.
word-spacing	Espacement entre les mots. Même remarque que pour letter-spacing.

L'espacement dans le corps du texte

La propriété white-space mérite tout de même une petite explication.

En temps normal, le *rendering* d'un contenu textuel retire tous les espacements (espaces, tabulations, retours chariot, etc.) au début et à la fin du texte, ramène toute autre série d'espacements à une seule espace classique (y compris les retours chariot), et va à la ligne quand c'est nécessaire (quand le texte arrive en bout de largeur du bloc conteneur).

On a donc trois comportements distincts : la réduction des espacements, le respect des retours chariot d'origine et le passage à la ligne pour honorer la largeur du conteneur (*wrapping*). Voici les définitions succinctes des valeurs possibles pour white-space :

Tableau B-14 Valeurs de white-space

Valeur	Réduction	Retours chariot	Wrapping
normal	Oui	Non	Oui
pre	Non	Oui	Non
nowrap	Oui	Non	Non
pre-wrap	Non	Oui	Oui
pre-line	Oui	Oui	Oui

Propriétés des tableaux

Tableau B-15 Propriétés des tableaux en CSS 2.1

Propriété	Description
<code>border-collapse</code>	Gère la fusion des bordures entre cellules adjacentes, et entre les cellules et la bordure du tableau. Désactivée par défaut (<code>separate</code>), peut être activée avec la valeur <code>collapse</code> . Je trouve ça beaucoup plus joli, personnellement...
<code>border-spacing</code>	Espacement entre bordures de cellules (si les bordures ne sont pas fusionnées). Peut contenir une ou deux tailles. Dans le second cas, distingue entre distances horizontale et verticale.
<code>caption-side</code>	Position du titre : au-dessus (<code>top</code> , défaut) ou en dessous (<code>bottom</code>).
<code>empty-cells</code>	Affiche ou masque les cellules vides. Par défaut, affiche (<code>show</code>). On les masque avec <code>hide</code> .
<code>table-layout</code>	Mode de calcul des largeurs du tableau et des cellules. Le mode par défaut, <code>auto</code> , est celui auquel on s'attend : il adapte les largeurs en fonction des contenus de celles-là. L'autre mode, <code>fixed</code> , utilise uniquement les spécifications de largeur pour le tableau, les colonnes, les bordures et l'espacement entre cellules. Il est plus rapide mais rend généralement moins bien.

Propriétés de l'interface utilisateur

Tableau B-16 Propriétés de l'interface utilisateur en CSS 2.1

Propriété	Description
<code>cursor</code>	Détermine l'aspect du curseur souris à utiliser lorsque celui-ci survole l'élément. Extrêmement utile en terme d'ergonomie. Les valeurs sont détaillées à la section 18.1 de la recommandation, mais je cite les principales : <code>auto</code> (défaut), <code>default</code> (curseur classique du système), <code>pointer</code> (comme pour un lien), <code>move</code> (idéal pour glisser-déplacer), <code>help</code> (idéal pour <code>abbr</code> et <code>acronym</code>).
<code>outline</code> -[<code>color</code> <code>style</code> <code>width</code>]	Affiche une délimitation autour d'un élément. Diffère d'une bordure en ce qu'elle n'occupe pas de place dans le modèle de boîte : elle est dessinée au-dessus du bord extérieur de la bordure. Elle ne comprend donc pas les marges. Peut être utile pour signaler qu'un élément est prêt à recevoir un dépôt lors d'un glisser-déplacer, mais encore mal prise en charge...

Pour aller plus loin...

Livres

CSS 2 – Pratique du design web

Raphaël Goetter

Eyrolles, juin 2005, 324 pages (de bonheur)

ISBN 2-212-11570-9

Le zen des CSS

Dave Shea

Eyrolles, novembre 2005, 296 pages

ISBN 2-212-11699-3

Cascading Style Sheets: The Definitive Guide

Eric Meyer

O'Reilly, novembre 2005, 508 pages

ISBN 0-596-00525-3

Mémento CSS

Raphaël Goetter

Eyrolles, novembre 2005, 14 pages (de mémento)

ISBN 2-212-11726-4

Sites

- La recommandation CSS 2.1, évidemment. Attention, seule la version anglaise a valeur de référence :
 - <http://www.w3.org/TR/CSS21/>
 - Version française de la 2.0 : <http://www.yoyodesign.org/doc/w3c/css2/cover.html>
- L'excellent site géré par Raphaël Goetter (jetez-vous sur son livre !), Alsa Créations : <http://www.alsacreations.com/>
- Le CSS Zen Garden, pour se convaincre qu'avec le même XHTML, on peut changer complètement de tête : <http://csszengarden.com/>
- A List Apart brille aussi en CSS : <http://alistapart.com/>
- Des styles décalés, les frontières de l'impossible : CSS Play.
<http://moronicbajebus.com/playground/cssplay/>
- Roger Johansson a plein de choses à vous raconter sur CSS, si vous allez au 456 Berea St. : <http://www.456bereastreet.com/>
- CSS Beauty : <http://www.cssbeauty.com/>

C

Le « plus » de l'expert : savoir lire une spécification

Il existe trois catégories de développeurs web. D'abord, ceux qui semblent toujours tout savoir, quitte à ne vous répondre que quelques instants plus tard, et qui expriment leur réponse avec un air d'autorité confiante dans l'exactitude de leur propos, laquelle se vérifie en effet à chaque fois. Ensuite ceux qui n'ont pas toutes les réponses, et semblent ne pas trop savoir où les chercher. Enfin, ceux qui manifestement n'ont qu'une compétence empirique : leurs pages « tombent en marche ».

Puisque vous avez ce livre entre les mains, on peut penser que vous souhaitez ne pas faire partie de la dernière catégorie, ni même avoir à travailler avec de telles personnes. Vous connaissez probablement un certain nombre de développeurs entrant dans la deuxième catégorie ; c'est peut-être d'ailleurs votre cas. Quant à ceux de la première catégorie, ces puits de connaissance apparemment sans fond, ils suscitent l'admiration de tous. Cette annexe vous propose modestement de tenter d'en faire partie.

De l'intérêt d'aller chercher l'information à la source

Il y a deux intérêts fondamentaux à être capable d'aller chercher l'information à la source. Le premier est parfaitement objectif et professionnel. Le second est plus subjectif et, comment dire... plus humain.

Certitude et précision

Les bonnes spécifications ont plusieurs qualités. Intrinsèquement d'abord, elles constituent le document de référence pour une technologie : elles font donc autorité sur la question. Utiliser correctement la technologie revient à l'utiliser conformément à la spécification. Si cela ne fonctionne pas alors que c'est exactement comme la spécification le demande, on sait que c'est notre environnement de travail qui est fautif et non notre code.

Bien sûr, cela peut simplement vouloir dire qu'on utilise du CSS 2.1 sur MSIE 6, auquel cas on ne peut pas laisser les choses telles quelles, il faudra trouver une solution.

Une bonne spécification est par ailleurs précise : elle doit indiquer tous les cas particuliers, toutes les nuances, tous les problèmes potentiels. Elle ne doit pas laisser de zone d'ombre. Généralement, cela signifie que la spécification est aride, ou en tout cas particulièrement verbeuse. Les excellentes spécifications arrivent à conjuguer une précision totale et une bonne lisibilité.

Quiconque maîtrise un sujet sur le bout des doigts le sait bien : il est très agréable de discuter de quelque chose qu'on connaît parfaitement. En particulier s'il s'agit d'aider quelqu'un à comprendre, à utiliser, à mettre au point. L'expertise est agréable. Être véritablement spécialiste d'un domaine précis et mettre cette expertise en œuvre est très agréable.

Savoir utiliser les spécifications d'une technologie procure ce que les moyens de deuxième main (livres, didacticiels, articles, ateliers, etc.) ne peuvent que difficilement donner : l'accès à une maîtrise totale, ou en tout cas l'accès à l'information totale.

« On m'a dit que là-dessus, c'est toi qui sais tout » : l'expertise

Il existe un deuxième avantage, plus humain celui-là. À force de faire preuve d'expertise sur un sujet donné, vous allez être connu pour cela. Un cercle toujours plus large de collègues et connaissances va faire l'association d'idées entre ce sujet et vous. Et de plus en plus, lorsqu'on aura besoin d'une information précise, pointue, fiable, on viendra vous voir.

Être un expert en XHTML, en balisage sémantique, en accessibilité, en CSS 2.1, en DOM niveau 2, en JavaScript et en Ajax ne vous apportera pas fortune et gloire (quoique...), mais dans votre travail, cela risque fort de vous apporter autre chose : *vous allez devenir indispensable.*

Rien que pour l'ego, c'est agréable. Mais cela peut aussi modifier vos préférences salariales, embellir votre CV et vous ouvrir de nouvelles opportunités.

Les principaux formats de spécifications web

Dans le cadre des technologies web, les spécifications utilisent essentiellement quatre formats.

Les recommandations du W3C

Les technologies gérées par le W3C sont publiées sous la forme de recommandations. On trouve deux abréviations courantes : TR (*Technical Report*) et REC (*Recommendation*). Il s'agit du statut finalisé d'une spécification, qui passe auparavant par plusieurs stades WD (*Working Draft*, ou ébauche).

La plupart des « langages descriptifs » du Web sont des technologies W3C. Citons principalement (X)HTML, XML, CSS, DOM, MathML, RDF, SMIL, SOAP, SVG, XPath et XSL/XSLT.

Les grammaires formelles de langages à balises : DTD et schémas XML

Les langages à balises disposent d'une grammaire formelle, très pratique pour retrouver rapidement le détail des attributs et éléments autorisés dans un contexte précis.

Suivant le cas (principalement selon l'origine et l'ancienneté du langage visé), la grammaire utilise soit une DTD (*Document Type Definition*), qui est un document SGML de syntaxe assez facile, soit un schéma XML, un document... XML, potentiellement plus puissant mais souvent très, très verbeux.

Par exemple, HTML et XHTML 1.0 utilisent des DTD, tandis que XHTML 1.1+, WSDL et XLink utilisent des schémas XML.

Il est à noter qu'un juste milieu existe au travers de la syntaxe Relax NG. Bien que celle-ci gagne chaque jour en popularité, elle n'est pas encore adoptée par les principaux organismes de standardisation, notamment le W3C.

Les RFC de l'IETF : protocoles et formats d'Internet

Enfin, la plupart des protocoles et formats de données du Web sont gérés et normalisés par l'IETF (*Internet Engineering Task Force*), un très large regroupement de professionnels qui est, véritablement, à l'origine d'Internet (premiers standards en 1969 !).

Les standards de l'IETF sont collectivement appelés les RFC (*Request For Comments*), et utilisent un format texte en 72 colonnes, très simple à lire. Il en existe plus de 4 600, dont plus de 300 ont vu le jour entre janvier et août 2006.

On y trouve notamment les spécifications pour HTTP, SMTP, POP, IMAP, FTP, Telnet, ICMP (la commande ping), SSL...

S'y retrouver dans une recommandation W3C

Commençons par explorer la structure d'une recommandation W3C. Le site officiel du W3C est <http://w3.org>. Vous y trouverez toutes les spécifications dans leur version anglaise, seule à être garantie : des traductions existent souvent, mais leur qualité n'est pas validée en détail par le W3C, même si ce dernier fournit un lien vers celles-ci depuis la version originale.

URL et raccourcis

Les recommandations ont souvent une URL assez longue, car elle contient quelques répertoires et surtout une date de version. Voici quelques exemples, qui donnent une idée des dégâts :

- <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>
- <http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/>
- <http://www.w3.org/TR/2002/CR-css-mobile-20020725>
- <http://www.w3.org/TR/2001/REC-xhtml11-20010531/>
- <http://www.w3.org/TR/2003/REC-SVG11-20030114/>
- <http://www.w3.org/TR/1999/REC-xpath-19991116>

Vous y observez le préfixe /TR/ en début de chemin, comme pour toutes les recommandations. On a ensuite l'année de parution, puis le chemin des documents de la spécification, qui démarre généralement par REC (on a ici aussi CR, pour *Candidate Recommendation*, dernier stade avant finalisation. On peut aussi trouver WD : *Working Draft*, ou NOTE, pour les documents à valeur informative).

Les chemins des spécifications précisent toujours la date exacte de parution du document à la fin, au format aaaammjj. On peut donc déduire, par exemple, que la dernière version du DOM niveau 2 HTML date du 19 janvier 2003, alors que celle du DOM niveau 2 noyau date du 13 novembre 2000.

Pour de nombreuses spécifications, il existe toutefois une URL « raccourcie », qui amène automatiquement à la dernière version. Voici les équivalents des URL mentionnées plus haut, avec quelques autres :

- <http://www.w3.org/TR/html401/>
- <http://www.w3.org/TR/xhtml1/>
- <http://www.w3.org/TR/CSS21/>
- <http://www.w3.org/TR/DOM-Level-2-Core/>
- <http://www.w3.org/TR/DOM-Level-2-HTML/>
- <http://www.w3.org/TR/css-mobile>
- <http://www.w3.org/TR/xhtml11/>
- <http://www.w3.org/TR/SVG11/>
- <http://www.w3.org/TR>xpath>

En fait, cela revient le plus souvent à supprimer l'année, le préfixe REC et la date en fin de nom. Notez qu'il y a parfois des *slashes* (/) terminaux, et parfois non. Dans certains cas (HTML 4.01, XHTML 1.0, CSS 2.1...) cela n'a aucune importance, dans d'autres vous obtiendrez une page intermédiaire.

Structure générale d'une recommandation

Une recommandation W3C a toujours la même structure générale.

D'abord l'en-tête :

- 1 titre avec version ;
- 2 statut (recommandation, ébauche...) et date de publication ;
- 3 liste de liens vers les formats disponibles (texte, HTML, PDF...) ;
- 4 liens vers la dernière version et la version précédente ;
- 5 liste des éditeurs, c'est-à-dire des responsables de la spécification.

La figure C-1 illustre l'en-tête de la recommandation DOM niveau 2 HTML.

Comme vous le voyez, la structure n'est pas toujours exactement identique à celle décrite plus haut, mais on retrouve très vite ses repères : ici, les formats sont simplement listés après la liste des éditeurs. Le lien vers les traductions, qui figure souvent après la partie introductory, se trouve ici en fin d'en-tête. Mis à part ceci, on reste dans le moule. Comparez avec l'en-tête de la recommandation pour HTML 4.01, qui correspond à l'ancienne façon de faire et reprend exactement notre liste (figure C-2).

Figure C-1
L'en-tête de la recommandation DOM niveau 2 HTML

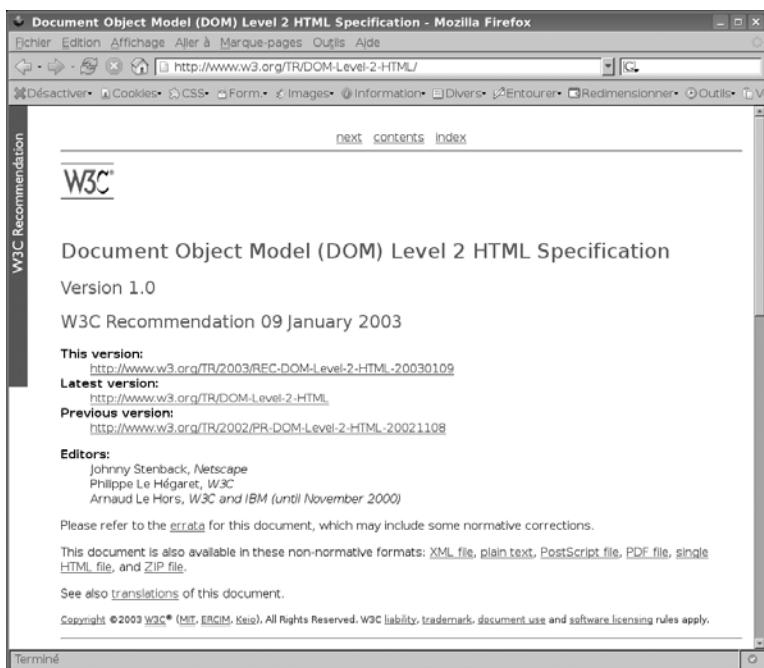
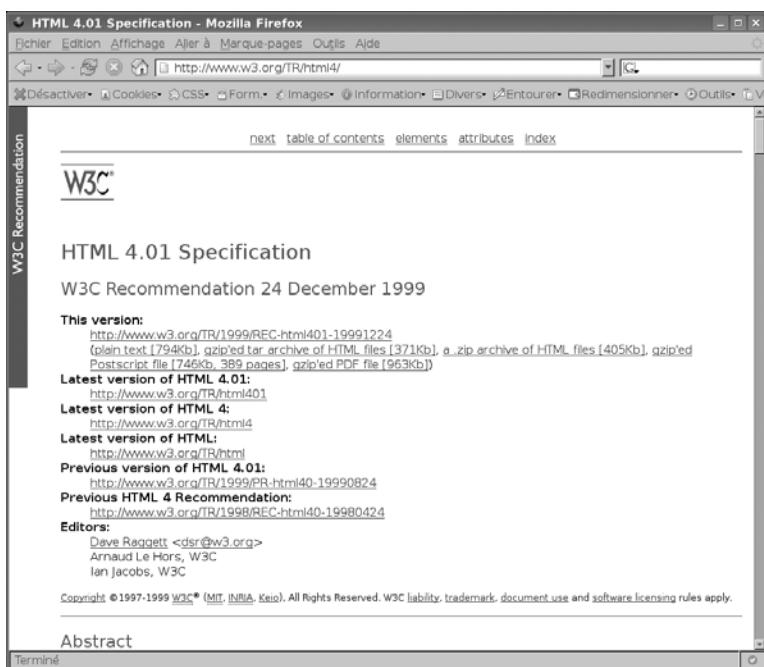


Figure C-2
L'en-tête de la recommandation HTML 4.01



Il a même une particularité, qui est de proposer les versions à jour et précédentes pour les variantes 4 et 4.01. Notez aussi que la liste des formats était bien moins lisible que la forme adoptée plus récemment.

On trouve ensuite la partie introductive, qui fournit :

- 1 *l'abstract*, qui décrit rapidement le rôle de la technologie spécifiée ;
- 2 le statut du document, qui contient toujours un texte plus ou moins pro forma sur le statut (recommandation, ébauche, etc.), l'état non normatif s'il s'agit d'une traduction, et fournit la liste des traductions connues (ou en tout cas un lien dessus, chercher le lien *translations* dans le corps du texte faute d'une section *Available languages*) ainsi qu'un lien vers les corrections ultérieures à la publication (*errata*) ;
- 3 la table des matières.

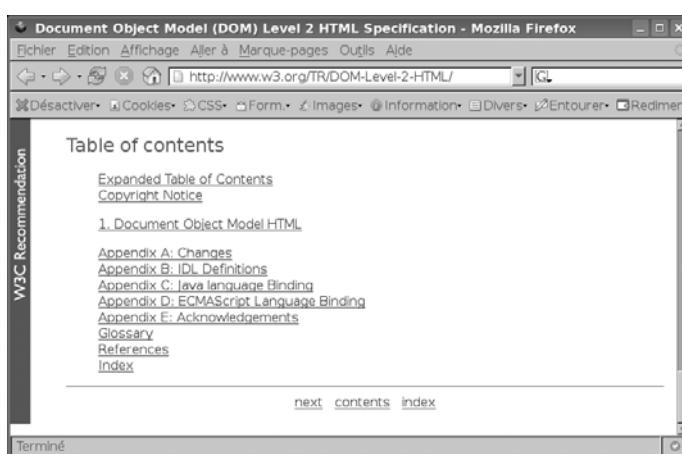
Suivant la spécification, on a alors plusieurs possibilités :

- Toute la spécification est sur la même page, le même document (cas de XHTML 1.0 ou XPath, par exemple).
- Seule la table des matières y figure et chaque section a une page dédiée (c'est le cas par exemple de HTML 4.01, XHTML 1.1 et CSS 2.1).
- La table des matières présentée ne référence que les parties incontournables (table des matières justement, copyright, annexes classiques, glossaire, références, index). Le cœur du sujet est exposé en une seule section (deux tout au plus), mais figure dans un sous-document (cas quasi systématique dans les spécifications du DOM, voir figure C-3).

On reconnaît vite la nature du document, rien qu'à la taille de l'ascenseur dans la barre de défilement verticale : s'il est très petit, il est probable qu'on ait toute la spécification sur une seule page !

Figure C-3

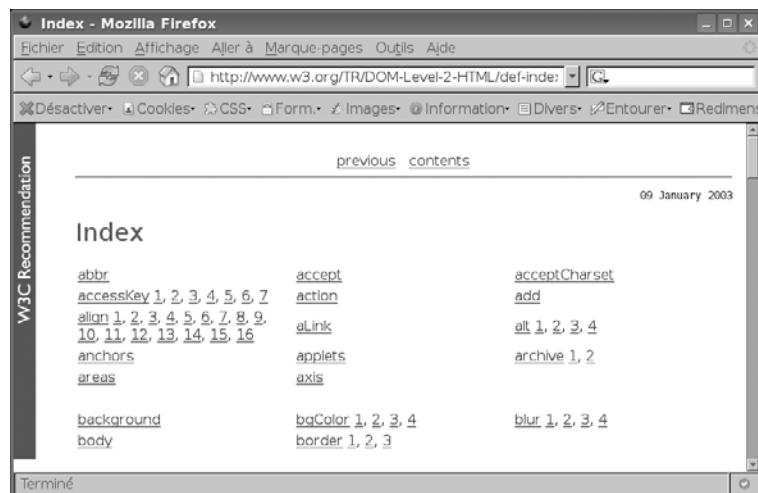
Spécifications DOM :
une table des matières
courte et un sous-document



Le format premier d'une recommandation est le HTML. Les documents utilisent donc abondamment les hyperliens, ce qui rend leur consultation plus pratique. Toutefois, la simple taille des recommandations fait qu'on s'y perd facilement. Pour s'y retrouver, on a deux moyens. Si la spécification ne comporte que quelques pages (voire une seule), l'outil de recherche intégré au navigateur doit permettre de s'y retrouver rapidement. On l'appelle généralement avec Ctrl+F.

Pour des spécifications plus distribuées, comportant de nombreuses pages, il est bon de regarder si la recommandation fournit un index, qui figure alors généralement tout en bas de la table des matières. Chaque terme important (notamment tous les noms d'éléments, de propriétés, de méthodes, d'interfaces, etc.) fait l'objet d'un lien ; en cas de liens multiples, le premier porte le nom de l'élément et les suivants des numéros.

Figure C-4
Index d'une recommandation W3C

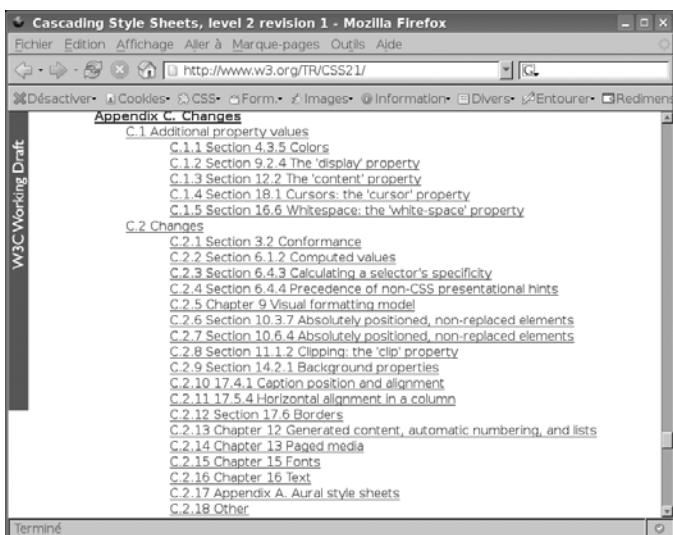


Astuce utile lorsqu'on cherche à déterminer si un aspect précis appartient à une version donnée ou à la précédente (ou simplement pour examiner rapidement en quoi la nouvelle version diffère de l'autre) : chaque recommandation dispose en annexe (et même dans les toutes premières annexes) d'une liste des changements. Suivant la taille de la recommandation, cette annexe est elle-même plus ou moins structurée. Ainsi, les changements pour CSS 2.1 (en réalité pour CSS 2.0 et 2.1) sont impressionnantes (figure C-5).

En revanche, dans le DOM niveau 2 HTML, on est plus sobre : une simple entrée comme annexe A, nommée *Changes*. Il faut dire que le document détaillant les changements depuis DOM niveau 1 pour les éléments relatifs à HTML fait à peine défiler deux écrans.

Figure C-5

Que de changements entre CSS 1 et CSS 2.1 !



Recours à des syntaxes formelles

Suivant ses besoins, une recommandation W3C va s'appuyer sur des syntaxes formelles adaptées pour décrire des aspects techniques. Les principales syntaxes employées sont :

- DTD pour les éléments de balisage jusqu'à XHTML 1.0 Strict ;
- Schéma XML à partir de XHTML 1.1 ;
- IDL (*Interface Description Language*) pour les interfaces (essentiellement dans le cadre du DOM) ;
- EBNF (*Extended Backus-Naur Form*), Lex ou Yacc pour le reste.

En raison de leur fréquence, nous étudierons plus en détail DTD et les schémas XML dans les sections qui suivront.

IDL est très facile à lire, car elle ressemble à une déclaration de classe abstraite ou d'interface dans les principaux langages objets. C'est parfois aussi simple que dans le code suivant.

Listing C-1 Déclaration IDL de l'interface HTMLElement (DOM niveau 2 HTML)

```
interface HTMLElement : Element {
    attribute DOMString id;
    attribute DOMString title;
    attribute DOMString lang;
    attribute DOMString dir;
    attribute DOMString className;
};
```

C'est parfois un peu plus compliqué, mais ça reste facilement lisible.

Listing C-2 Déclaration IDL de l'interface `HTMLSelectElement`

```
interface HTMLSelectElement : HTMLElement {
    readonly attribute DOMString          type;
    attribute long                      selectedIndex;
    attribute DOMString          value;
    // Modified in DOM Level 2:
    attribute unsigned long   length;
    // raises(DOMException) on setting

    readonly attribute HTMLFormElement form;
    // Modified in DOM Level 2:
    readonly attribute HTMLOptionsCollection options;
        attribute boolean      disabled;
        attribute boolean      multiple;
        attribute DOMString    name;
        attribute long         size;
        attribute long         tabIndex;
    void           add(in HTMLElement element,
                      in HTMLElement before)
                      raises(DOMException);
    void           remove(in long index);
    void           blur();
    void           focus();
};
```

Quant à EBNF, il s'agit d'une des plus anciennes syntaxes textuelles de description de grammaire. Vous trouverez quelques explications sur cette syntaxe, plutôt simple, aux deux URL suivantes :

- <http://developpeur.journaldunet.com/tutoriel/theo/050831-notation-bnf-ebnf.shtml>
- <http://fr.wikipedia.org/wiki/EBNF>

Certaines spécifications utilisent des descriptions reposant plutôt sur les syntaxes Lex (ou Flex) et Yacc (ou Bison), bien connues des développeurs C. Ce n'est pas très éloigné d'EBNF. Voici un exemple tiré de la recommandation CSS 2.1, qui repose lui-même sur quelques définitions établies plus haut dans le document.

Listing C-3 Définition Lex de syntaxe générale pour une feuille de styles CSS

```
stylesheet  : [ CDO | CDC | S | statement ]*;
statement   : ruleset | at-rule;
at-rule     : ATKEYWORD S* any* [ block | ';' S* ];
block       : '{' S* [ any | block | ATKEYWORD S* | ';' S* ]* '}' S*;
ruleset    : selector? '{' S* declaration? [ ';' S* declaration? ]* '}' S*;
```

```

selector      : any+;
declaration  : DELIM? property S* ':' S* value;
property     : IDENT;
value        : [ any | block | ATKEYWORD S* ]+;
any          : [ IDENT | NUMBER | PERCENTAGE | DIMENSION | STRING
               | DELIM | URI | HASH | UNICODE-RANGE | INCLUDES
               | DASHMATCH | FUNCTION S* any* ')';
               | '(' S* any* ')' | '[' S* any* ']' ] S*;
```

Il faut bien comprendre que, l'immense majorité du temps, les descriptions EBNF, Lex ou Yacc visent plus les développeurs de logiciels implémentant la technologie que ceux qui *utilisent* cette même technologie. Le texte environnant fournit généralement tous les détails nécessaires pour une utilisation courante.

Les descriptions de propriétés CSS

La section « Déchiffrer une DTD », plus loin dans cette annexe, montrera comment lire et exploiter les fragments de DTD employés pour décrire les langages à balises, par exemple HTML. Avant d'en finir avec les recommandations W3C, je voudrais tout de même donner quelques informations supplémentaires sur deux formats très consultés dans la pratique : celui décrivant les propriétés CSS, et celui décrivant les propriétés et méthodes du DOM.

Commençons donc par les propriétés CSS. Par souci de référence, nous utiliserons la version originale, en anglais. Prenons par exemple la description de la propriété `white-space`, dans la section *Text*. La spécification utilise la représentation suivante pour décrire ladite propriété :

'white-space'	
<i>Value:</i>	normal pre nowrap pre-wrap pre-line inherit
<i>Initial:</i>	normal
<i>Applies to:</i>	all elements
<i>Inherited:</i>	yes
<i>Percentages:</i>	N/A
<i>Media:</i>	visual
<i>Computed values:</i>	as specified

- 1 *Value* détaille la liste des valeurs possibles, séparées par des *pipes* (|), symbole fréquent pour indiquer une alternative. Les valeurs exprimées sont normatives et la casse est parfois significative. On a ici 6 valeurs possibles, dont l'incontournable `inherit` pour une propriété héritable.

- 2 *Initial* décrit la valeur par défaut, *none* s'il n'y en a aucune.
- 3 *Applies to* décrit les catégories d'éléments qui disposent de cette propriété. Ici elle s'applique à tous, mais des valeurs courantes sont *block-level elements*, *inline elements*, etc. Il s'agit des catégories déterminées par le modèle visuel CSS, évoqué à l'annexe B.
- 4 *Inherited* indique si la propriété est héritable ou pas ; une propriété héritable prend sa valeur par défaut dans son élément conteneur : c'est le principe de la cascade. Toutes les propriétés ne sont pas héritables (par exemple, `text-decoration` ne l'est pas).
- 5 *Percentages* est utile quand les valeurs possibles incluent une notation en pourcentage. C'est principalement le cas des tailles (de boîte ou de police de caractères). Cette valeur indique alors à quoi se réfèrent les pourcentages (par exemple, la largeur du bloc conteneur).
- 6 *Media* indique le ou les média CSS pour lesquels la propriété a du sens. Ici on est sur `visual`, ce qui couvre tous les média visuels : `screen`, `print`, `projection`, etc., tout en excluant par exemple les média comme `aural` (lecteurs d'écran).
- 7 *Computed values*, enfin, précise les ajustements à apporter à la valeur en fonction du contexte. C'est parfois simple et donc indiqué à la volée. Quand c'est plus complexe, on a généralement *as specified*, et les détails dans le texte qui suit. Par exemple, la valeur de la propriété `text-align` varie suivant celle de la propriété `white-space`.

Les descriptions de propriétés et méthodes DOM

Autre format fréquemment consulté, les propriétés et méthodes DOM. On a déjà illustré le code IDL utilisé pour représenter une interface DOM complète, mais ces fragments de code sont suivis d'explications plus détaillées, bien entendu. Une interface DOM est toujours spécifiée en plusieurs morceaux :

- 1 le code IDL de l'interface ;
- 2 les descriptions des attributs éventuels ;
- 3 les descriptions des méthodes éventuelles.

Le code IDL fournit quelques informations précieuses. D'abord, au niveau de la déclaration de l'interface elle-même, si celle-ci étend une autre interface, on le voit immédiatement. Quelques exemples :

```
interface HTMLDocument : Document {  
    ...  
}  
interface HTMLElement : Element {  
    ...  
}  
interface HTMLImageElement : HTMLElement {  
    ...  
}
```

On voit clairement qu'un élément `img`, qui expose naturellement l'interface `HTMLImageElement`, expose donc aussi l'interface `HTMLElement`, et donc `Element`, et donc `Node`. Il fournit alors de nombreuses propriétés et méthodes !

Par ailleurs, le code IDL fournit une vue globale sur les types des propriétés et ceux des méthodes (types de retour, types des arguments). Quelques exemples là aussi, au travers du code IDL de `HTMLOptionsCollection`, reformulé :

```
// Introduced in DOM Level 2:
interface HTMLOptionsCollection {
    attribute unsigned long length;
    // raises(DOMException) on setting
    Node item(in unsigned long index);
    Node namedItem(in DOMString name);
};
```

Que trouve-t-on ici ? D'abord, les attributs sont reconnaissables à ce qu'ils sont préfixés par `attribute`, et n'ont pas de parenthèses après leur nom. Le mot réservé `attribute` est parfois précédé de `readonly`, ce qui est très important : cela signifie qu'il est en lecture seule. Toute tentative d'écriture générera sans doute une exception.

Entre `attribute` et le nom de l'attribut, on a le type. IDL utilise des types primitifs issus du C, et des types objets qui sont soit d'autres interfaces du DOM, soit des types classiques définis par le DOM noyau.

Les méthodes n'ont pas `attribute` au début, mais un type de retour, un nom, et des parenthèses entre lesquelles on peut lister des paramètres, avec leur sens, leur type et leur nom. Toutes ces informations sont précieuses, mais le sens est généralement `in` et peut être ignoré. Il signifie simplement que la méthode utilise l'argument sans le modifier : c'est un paramètre local, passé par valeur, si vous préférez.

Voici une liste des principaux types utilisés dans l'IDL des spécifications DOM.

Tableau C-1 Principaux types utilisés dans l'IDL du DOM

Type	Description
[<code>unsigned</code>] <code>short</code>	Nombre entier sur 16 bits. Soit signé (-32 768 à 32 767), soit non signé (0 à 65 535).
[<code>unsigned</code>] <code>long</code>	Nombre entier sur 32 bits. Même remarque pour le signe. C'est le type numérique le plus courant.
<code>boolean</code>	Valeur booléenne : <code>false</code> ou <code>true</code> .
<code>void</code>	Aucune valeur. Type de retour des méthodes ne renvoyant rien.
<code>DOMString</code>	Chaîne de caractères Unicode (sur 16 bits).
<code>DOMTimeStamp</code>	Nombre entier positif sur 64 bits représentant un nombre de millisecondes depuis le début de l'ère (1er janvier 1970 00:00:00 GMT). De nombreux langages représentent ainsi les dates.

Tableau C-1 Principaux types utilisés dans l'IDL du DOM (suite)

Type	Description
DOMException	Le type standard des exceptions levées par les méthodes. Contient simplement une propriété numérique entière nommée <code>code</code> , qui vaut l'une des constantes <code>xxx_ERR</code> décrites dans les recommandations du DOM.
Interface DOM (ex. Node)	Eh bien, l'interface en question... Reportez-vous à sa définition !

Les valeurs numériques signées sont assez rares : on utilise principalement `unsigned long`.

Après le code IDL, on trouve une section *Attributes* s'il y a des attributs, et une section *Methods*... s'il y a des méthodes.

Une section d'attribut reprend son nom, son type et son éventuelle contrainte de lecture seule. Un texte décrit l'attribut plus en détail, et si celui-ci a des contraintes pour son écriture, un paragraphe particulier précise l'information *Exception on setting* qui figurait déjà, normalement, en commentaire dans le code IDL.

Une section de méthode reprend son nom, puis décrit la méthode plus en détail avec un texte explicatif. Ce texte est obligatoirement suivi de trois sections : les paramètres, la valeur de retour et les exceptions potentielles. Les méthodes simplissimes se retrouvent donc avec trois embryons de sections. Ce qui donne par exemple :

close

Closes a document stream opened by `open()` and forces rendering.

No Parameters

No Return Value

No Exceptions

Ces sections fournissent souvent le petit détail qui explique pourquoi votre appel produit une erreur, ou ne donne rien, ou encore engendre un avertissement. Bien les lire est précieux.

Quand vous utilisez pour la première fois une interface DOM, lisez sa documentation complète. Ne vous contentez pas de quelques bouts qui semblent répondre à vos questions les plus immédiates. Prenez quelques minutes de plus pour lire l'ensemble, cela vous économisera souvent bien des heures de débogage par la suite.

Déchiffrer une DTD

La DTD (*Document Type Definition*) constitue le premier format historique de grammaire pour les langages à balises. Une DTD est rédigée en SGML (*Standard Generalized Markup Language*), langage dont le HTML est en quelque sorte une réalisation.

Si vous rédigez correctement vos pages web, vous faites référence à une DTD tout au début de votre document, à l'aide d'une instruction DOCTYPE. Par exemple, la première ligne de votre page web dit normalement :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"  
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Comment ça, « non » ?! Mais c'est mal ! Vous ne faites pas du XHTML 1.0 Strict ? Vous n'avez pas honte ? Allez donc relire l'annexe A et faites pénitence.

Déclarer explicitement sa DTD de référence dans une page web a un double avantage. D'une part, cela permet d'utiliser correctement un validateur HTML : celui-ci pourra détecter votre grammaire au lieu d'essayer de la deviner, et vous assurera ainsi une vérification de grammaire appropriée. D'autre part, déclarer une DTD stricte (qu'elle soit HTML 4.01 ou XHTML 1.0) permet à votre navigateur de passer en mode « respect des standards », notamment en ce qui concerne les CSS. Pour plus de détails, cherchez donc l'expression *doctype switching* sur Google.

Mais revenons à nos DTD dans le cadre de spécifications. Une DTD constitue une spécification formelle pour un langage à balises (par exemple HTML 4.01). Elle n'explique pas le sens des balises ou des attributs, mais décrit quelles balises et quels attributs sont disponibles, où et quand.

Les spécifications HTML et apparentées ont pris l'habitude, au début de chaque section décrivant un élément, de présenter le fragment correspondant de la DTD. Observez par exemple la spécification de HTML 4.01 (sur laquelle repose XHTML 1.0, ne l'oubliez pas) pour l'élément form.

Listing C-4 Le fragment de la DTD HTML 4.01 pour form

```
<!ELEMENT FORM - - (%block;|SCRIPT)+ -(FORM) -- interactive form -->  
<!ATTLIST FORM  
    %attrs;                      -- %coreattrs, %i18n, %events --  
    action    %URI;               #REQUIRED -- server-side form handler --  
    method    (GET|POST)          GET     -- HTTP method used to submit the form --  
    enctype   %ContentType;      "application/x-www-form-urlencoded"  
    accept    %ContentTypes;     #IMPLIED -- list of MIME types for file upload --  
    name      CDATA              #IMPLIED -- name of form for scripting --  
    onsubmit  %Script;           #IMPLIED -- the form was submitted --  
    onreset   %Script;           #IMPLIED -- the form was reset --  
    accept-charset %Charsets;    #IMPLIED -- list of supported charsets --  
>
```

On a ici la description de l'élément `form` lui-même, avec son nom et la description de son contenu. Vous remarquez sans doute qu'ici, les noms d'éléments sont en majuscules, « à l'ancienne ». C'était la norme du temps de HTML (au siècle dernier, donc), mais depuis que XHTML est arrivé, on est sensible à la casse et les nouvelles normes officialisent la casse minuscule.

Les recommandations suivent d'ailleurs les pratiques de leur temps : même s'il est aujourd'hui communément admis que XHTML est incontournable, et qu'il faut donc respecter les règles de fermeture de balises, de valeurs d'attribut entre guillemets, et de balises en minuscules, la recommandation HTML foisonne toujours d'exemples « d'époque », pour ainsi dire, où beaucoup d'éléments ne sont pas fermés, sont en majuscules et ne protègent pas leurs valeurs d'attribut !

Détaillons rapidement la syntaxe de notre fragment. On a d'abord :

```
| <!ELEMENT FORM -- (%block;|SCRIPT)+ -(FORM) -- interactive form -->
```

Le début, `<!ELEMENT`, indique qu'on définit un élément. Le nom suit : `FORM`.

Après les deux tirets séparés, on trouve une description du modèle de contenu : `(%block;|SCRIPT)+ -(FORM)`. On peut traduire ce modèle comme ceci : « soit le modèle *block*, soit l'élément *SCRIPT*, le tout autant de fois qu'on veut (mais au moins une fois), en revanche, pas d'élément *FORM* imbriqué ».

En effet, `%block` est ce qu'on appelle une référence d'entité, sur laquelle il est d'ailleurs possible de cliquer dans la recommandation. La définition de l'entité `%block` la décrit comme pouvant être réalisée par un certain nombre de balises connues, dont par exemple `P`, `H1`, `UL`, `PRE`, `DIV`, mais aussi `FORM`, ce qui amène justement l'exclusion explicite dans notre définition d'élément `FORM`.

La dernière partie, `-- interactive form --`, constitue un commentaire de fin de définition, qui nous signale que l'élément `FORM` permet de réaliser un formulaire utilisateur.

Passons maintenant à la liste des attributs pour l'élément `FORM`. Classiquement, elle figure dans la DTD juste après la définition de l'élément lui-même. J'ai retiré les alignements entre les champs pour améliorer la lisibilité individuelle des extraits. La liste des attributs débute par :

```
| <!ATTLIST FORM
  %attrs; -- %coreattrs, %i18n, %events --
```

On déclare ici une définition d'attributs pour l'élément `FORM`. Les premiers attributs sont référencés par l'entité `%attrs`, dont le commentaire nous apprend qu'elle représente une combinaison des entités `%coreattrs`, `%i18n` et `%events`.

Ces entités regroupent les attributs dits noyau (`id`, `class`, `style` et `title`), ceux relatifs à la gestion des langues (`lang` et `dir`) et ceux relatifs aux événements JavaScript (par exemple `onclick`). Vous n'utiliserez bien entendu jamais ces derniers, puisque vous faites de l'*unobtrusive JavaScript*, n'est-ce pas ?

Que dit la suite ?

```
| action %URI; #REQUIRED -- server-side form handler --
```

On a là un attribut `action`, dont le modèle de contenu est référencé par l'entité `%URI` (laquelle, idéalement, devrait décrire la syntaxe d'un URI, mais la syntaxe DTD étant pauvre, elle se contente, faute de mieux, de dire simplement « texte quelconque »....). On précise aussi que cet attribut est obligatoire (et je me suis rendu coupable d'infraction à cette règle dans certains exemples JavaScript de ce livre, je l'avoue...).

Deux autres exemples intéressants de définition d'attributs :

```
| method (GET|POST) GET -- HTTP method used to... --
| name      CDATA          #IMPLIED -- name of form for scripting --
```

On a ici une autre forme de modèle de contenu, qui dit en substance : « l'attribut `method` peut valoir `GET` ou `POST`, mais pas autre chose » (ce qui embête tant aujourd'hui les partisans d'une approche REST pour les API web). On voit aussi une valeur par défaut au lieu de `#REQUIRED` : si la méthode n'est pas précisée, elle vaudra `GET`.

La définition de l'attribut `name` indique qu'il s'agit d'un texte quelconque (type spécial `CDATA`, pour *character data*), et qu'il est optionnel sans valeur par défaut (`#IMPLIED`).

Voilà une introduction suffisante. Astuce intéressante : la spécification HTML 4.01 a jugé bon de fournir une présentation plutôt détaillée des syntaxes DTD utilisées, et cela directement dans la recommandation ! Voici l'adresse de la version française afin de vous faciliter le plus possible l'acquisition : <http://www.la-grange.net/w3c/html4.01/intro/sgmltut.html#h-3.3>. Si cela ne suffit pas, vous avez une introduction générique assez bien faite qui pourra peut-être mieux vous satisfaire sur <http://www.w3schools.com/dtd/default.asp>.

J'ai dit tout à l'heure que la DTD précisait quelles balises et quels attributs étaient disponibles, où et quand. Dit comme cela, on a l'impression que toute l'information peut être transmise par la DTD seule, et que le corps du texte de la recommandation n'est là que pour préciser le sens des éléments et des balises et ajouter quelques informations de contexte.

En réalité, une DTD n'est pas très précise. La syntaxe disponible ne permet pas de préciser de nombreux cas de figure courants, qu'il faut alors décrire dans le corps du texte. On ne peut pas formuler certaines règles simples, par exemple indiquer qu'un

élément A peut avoir au maximum 5 éléments fils B, sans même parler d'indiquer une fourchette allant de 2 à 5 éléments.

On ne peut pas non plus exprimer des exclusions entre les attributs, ou indiquer que si tel attribut est présent, tel autre doit l'être aussi. Également, on ne peut pas indiquer que les valeurs de tel attribut, toutes balises comprises, doivent être uniques dans le document (cas flagrant, par exemple en HTML : l'attribut `id`). Bref, les DTD sont limitées.

C'est pourquoi la communauté s'est tournée vers une autre syntaxe. Hélas, on a alors poussé jusqu'à l'extrême inverse : pour pouvoir tout décrire, on a créé une syntaxe si verbeuse, que la plupart des cas simples prennent de nombreuses lignes à représenter. Ce sont les schémas XML.

Naviguer dans un schéma XML

Les schémas XML ont pris la relève des DTD pour décrire formellement les possibilités de combinaison et d'inclusion d'éléments et d'attributs dans les langages à balises. Et quand on sait combien le W3C s'est épris des technologies XML (plus de vingt à ce jour, de XML lui-même à XSLT, en passant par XPath, XML Query, SVG ou encore MathML...), on imagine facilement quel usage intensif est aujourd'hui fait des schémas XML.

Un schéma XML est lui-même un document XML. La syntaxe a été conçue pour couvrir tous les besoins imaginables. Et comme souvent dans de tels cas, elle aboutit à quelque chose de très épais pour les cas simples...

Prenons par exemple un fragment du schéma pour XHTML 1.1 Strict.

Listing C-5 Le schéma XML de l'élément form en XHTML 1.1

```
<xs:attributeGroup name="xhtml.form.attlist">
    <xs:attributeGroup ref="xhtml.Common.attrib"/>
    <xs:attribute name="action" type="xh11d:URI" use="required"/>
    <xs:attribute name="method" default="get">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="get"/>
                <xs:enumeration value="post"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="enctype" type="xh11d:ContentType"
        default="application/x-www-form-urlencoded"/>
    <xs:attribute name="accept-charset" type="xh11d:Charsets"/>
    <xs:attribute name="accept" type="xh11d:ContentTypes"/>
```

```
</xs:attributeGroup>
<xs:group name="xhtml.form.content">
    <xs:sequence>
        <xs:choice maxOccurs="unbounded">
            <xs:group ref="xhtml.B1kNoForm.mix"/>
            <xs:element name="fieldset" type="xhtml.fieldset.type"/>
        </xs:choice>
    </xs:sequence>
</xs:group>
<xs:complexType name="xhtml.form.type">
    <xs:group ref="xhtml.form.content"/>
    <xs:attributeGroup ref="xhtml.form.attlist"/>
</xs:complexType>
```

À qui aura le front de s'insurger : « c'est illisible ! », « c'est n'importe quoi ! », ou encore « beaucoup de bruit pour rien ! », je répondrai d'un air affable « vous avez bien raison ».

Seulement voilà, le W3C s'est amouraché des schémas XML et il n'est pas le seul, loin de là ! L'univers J2EE par exemple a basculé des DTD vers les schémas autour de l'apparition des Servlets 2.4 et de JSP 2.0, il y a déjà longtemps.

Parmi les concepts clefs d'un schéma XML, il faut retenir qu'on définit le plus souvent séparément un type (une grappe d'éléments et de balises) et les noms des éléments qui reposent sur ce type (ce qui permet certes une factorisation fort agréable). Les structures simples sont déclarées à l'aide de balises `<xs:simpleType>`, et les types complexes (en réalité, presque tous) au moyen de `<xs:complexType>`.

La notion de *groupe*, qui permet d'exprimer une exclusivité d'éléments ou d'attributs, ou de décrire des dépendances, est représentée par `<xs:group>` pour les éléments et `<xs:attributeGroup>` pour les attributs. Les éléments et attributs sont désignés respectivement par `<xs:element>` et `<xs:attribute>`, les contraintes exprimées soit avec des attributs d'occurrence (`minOccurs` et `maxOccurs`), soit avec l'attribut `use`, ou encore avec des descriptions complexes `<xs:restriction>`. Le fragment présenté en listing C-5 est sympathique, parce qu'il illustre au moins un cas possible pour presque toutes ces syntaxes.

Il me faudrait un ouvrage entier pour vous présenter toutes les possibilités des schémas XML, mais si le sujet vous intéresse, vous trouverez une présentation bien faite et assez détaillée, pas à pas, sur http://www.w3schools.com/schema/schema_intro.asp.

Pour terminer, sachez qu'une syntaxe alternative fait de plus en plus d'aficionados, car tout en étant basée elle aussi sur XML, et bien que permettant d'exprimer tout ce qui est exprimable en schémas XML, elle est incomparablement plus simple et plus lisible. Il s'agit de Relax NG (nouvelle génération).

Relax NG est suffisamment populaire pour que la plupart des bibliothèques de traitement de grammaires XML la prennent aujourd’hui en charge (qu’on travaille en Java, C#, Delphi ou Ruby...). Toutefois, elle n’a pas encore su gagner le cœur des grands organismes de standardisation. Pour vous faire une idée, consacrez donc quelques minutes à ce didacticiel sur son site officiel, qui permet de bien cerner la question : <http://relaxng.org/tutorial-20011203.html>. En guise de *teaser*, voici l’équivalent Relax NG du listing C-5.

Listing C-6 La spécification Relax NG de l’élément form en XHTML 1.1

```
<define name="form">
  <element name="form">
    <ref name="form.attlist"/>
    <oneOrMore>
      <ref name="Block.class"/>
    </oneOrMore>
  </element>
</define>

<define name="form.attlist">
  <ref name="Common.attrib"/>
  <attribute name="action">
    <ref name="URI.datatype"/>
  </attribute>
  <optional>
    <attribute name="method">
      <choice>
        <value>get</value>
        <value>post</value>
      </choice>
    </attribute>
  </optional>
  <optional>
    <attribute name="enctype">
      <ref name="ContentType.datatype"/>
    </attribute>
  </optional>
</define>
```

L’exemple étant bien choisi, on ne voit pas une immense différence de taille (28 lignes dans les deux cas), mais sentez-vous combien la seconde version est plus lisible que la première ?

Parcourir une RFC

Une RFC (*Request For Comments*) est un standard Internet élaboré par l'IETF (*Internet Engineering Task Force*). Je dis bien « Internet » et non « Web », car alors que le Web a vu le jour en 1992 avec la première page HTML, sous la houlette de Tim Berners-Lee, Internet existe, lui, depuis 1969 (même s'il s'appelait alors ARPANET).

Ce sont aujourd’hui des centaines de protocoles et de formats qui font fonctionner ce réseau mondial. Le grand public connaît les plus visibles : HTTP, POP, IMAP, SMTP, SSL, TLS, FTP... Ceux qui prêtent attention aux détails de leurs clients de messagerie connaissent peut-être aussi MIME. Il faut sans doute être dans l’informatique pour en connaître d’autres, comme SNMP, NTP, SSH, Telnet, RTSP et bien d’autres. Sans parler des formats, de CSV à Atom.

Au total, ce sont plus de 4 600 documents standardisés qui ont déjà été émis, et avec environ 500 documents par an ces derniers temps, le phénomène ne fait qu'accélérer.

Format général d'une RFC

Le format d'une RFC a toutefois perduré à travers les âges et trahit aujourd’hui ses origines très modestes, à une époque où les imprimantes matricielles 8 points étaient le dernier cri, et où la notion même de réseau d’ordinateurs n’était encore qu’une vision pleine d’espoir.

Le format principal d'une RFC est un fichier texte utilisant le jeu de caractères US-ASCII sur 7 bits (pas d’accents, pas de signes diacritiques, etc.). Le texte est formaté à 72 caractères de large. Certains termes ont un sens particulier, par exemple *must*, *should*, *can*, *must not*. Leur usage est défini par la RFC 2219. Une RFC est obligatoirement en anglais et le fichier porte l’extension .txt. On a encore bien des règles.

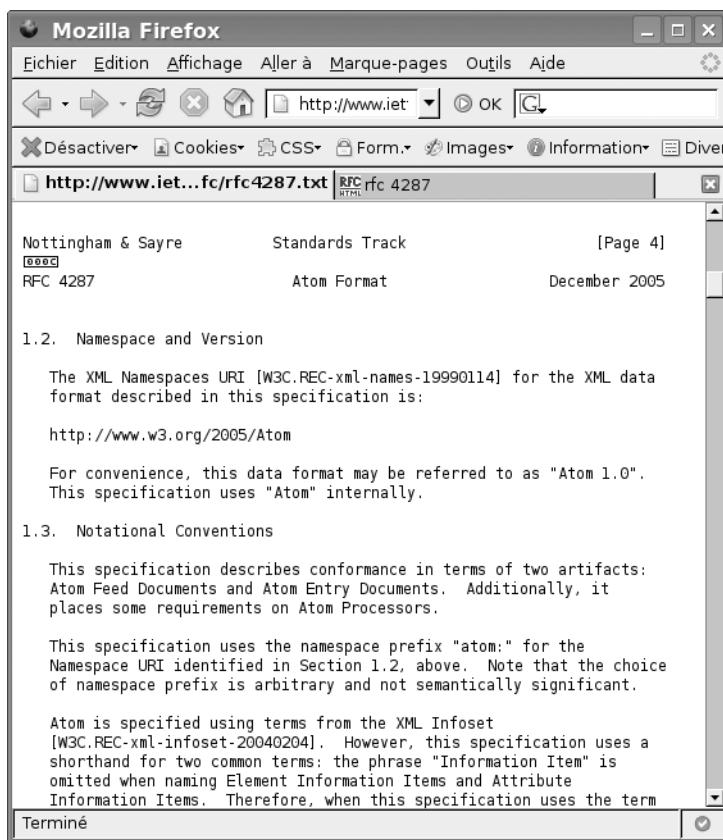
La RFC 2223 donne des détails sur le format ; une ébauche en plan depuis 2004 après 8 révisions, la 2223bis, met à jour certains points. On trouve même une RFC décrivant comment configurer MS Word pour produire un fichier RFC valide : la RFC 3285 !

Sachez par ailleurs qu'il existe trois sous-catégories de RFC : les standards de premier plan (STD), les pratiques recommandées (BCP, *Best Current Practice*) et les notes informatives (FYI, *For Your Information*). Dernier point : les RFC publiées le 1^{er} avril sont systématiquement des canulars. Faites donc une recherche, ça vaut le détour.

Enfin, le format textuel d'une RFC rend difficile la navigation à l'intérieur du document, les références n'étant pas des hyperliens. Aussi, sachez qu'il existe un accès HTML aux documents, dont la navigation est par conséquent facilitée. Pour obtenir la version HTML d'une RFC, c'est facile : prenez d'abord l'URL officielle de la version principale, par exemple : <http://www.ietf.org/rfc/rfc4287.txt>.

Figure C-6

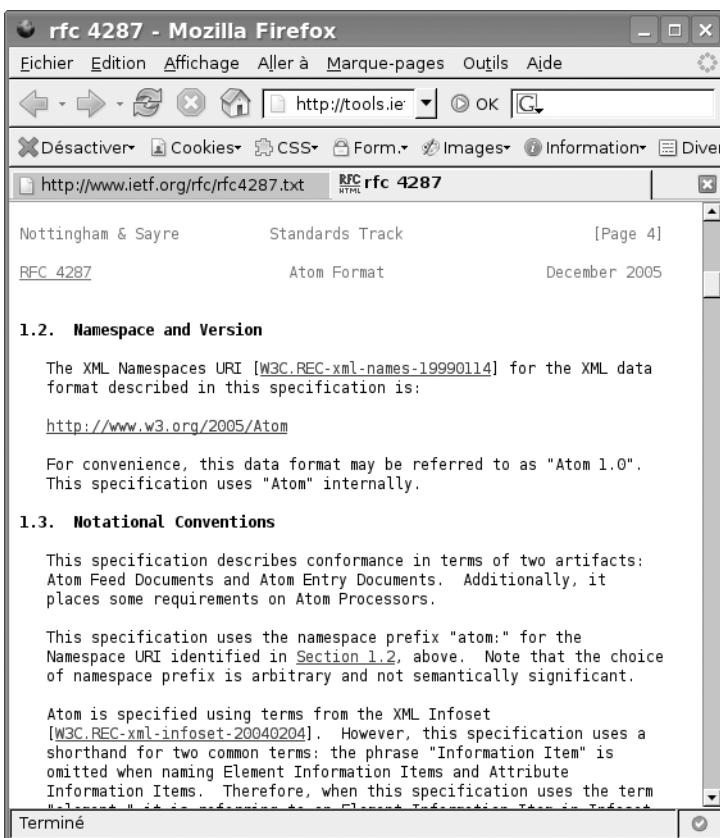
L'aspect texte brut de la RFC du format Atom



Remplacez ensuite le « www. » par « tools. », le chemin /rfc/ par /html/ et retirez l'extension. Vous obtenez ceci : <http://tools.ietf.org/html/rfc4287>.

Et voilà une version HTML de votre RFC, avec des liens automatiques pour les numéros de page, les URL, les références et les renvois entre sections. Par ailleurs, certaines portions sont mises en gras (titres) ou en italique, et les en-têtes et pieds de page sont grisés. Jugez plutôt (figure C-7).

Figure C-7
La RFC passée
à la moulinette HTML



Structure générale d'une RFC

Une RFC commence toujours par son en-tête déclaratif, repris en en-tête de chaque page. Voici un exemple, raccourci en largeur pour tenir sur cette page :

Network Working Group
Request for Comments: 4287
Category: Standards Track

M. Nottingham, Ed.
R. Sayre, Ed.
December 2005

On a sur la gauche le nom du groupe de travail (l'IETF en compte un certain nombre), le statut (RFC) et le numéro du document. Les RFC sont en effet le plus souvent référencées par leur numéro. La catégorie indique plus précisément le statut. Ici, le terme *Standards Track* confirme que le document a valeur de standard.

Sur la droite, on trouve la liste des auteurs principaux (le « Ed. » signifie *Editor*) et la date de publication du document, avec le mois et l'année.

L'en-tête des pages suivantes reprendra toutes ces informations ainsi que le numéro de page et le titre du document :

Nottingham & Sayre

Standards Track

[Page 1]

RFC 4287

Atom Format

December 2005

Une RFC comporte obligatoirement une introduction, dont les premiers paragraphes doivent décrire avec concision le contexte et l'objectif du standard. L'IETF est ici généralement plus efficace que le W3C dans ses résumés...

L'introduction comprend obligatoirement, généralement vers la fin, une section du type *Notational Conventions*, qui détaille les notations propres au document, et précise systématiquement le sens formel de termes comme *must*, *should*, *cannot*, etc. en faisant une référence à leur définition exacte dans la RFC 2119.

Une RFC se clôt généralement par une liste de références normatives (autres standards) ou informatives, une liste des contributeurs au standard (auteurs n'ayant pas un statut « principal »), et souvent des versions consolidées de grammaires formelles ou autres informations techniques fragmentées dans le corps du document.

Il n'est pas rare de voir, dans les derniers chapitres d'une RFC, un *IANA Considerations*, si le standard entraîne le dépôt de nouveaux types MIME ou réquisitionne certains numéros de ports réseau, ainsi qu'un *Security Considerations*, qui détaille toute faille de sécurité potentielle envisagée par les auteurs.

Enfin, les RFC font une utilisation intensive de la syntaxe EBNF, évoquée plus haut. Prenez par exemple la RFC 2616 (<http://tools.ietf.org/html/rfc2616>), celle qui décrit HTTP/1.1. La syntaxe des en-têtes de requête et de réponse est décrite en EBNF. En voici un extrait dans le listing suivant.

Listing C-7 Syntaxe EBNF de l'en-tête de requête Accept en HTTP/1.1

```

Accept      = "Accept" ":"  
            #( media-range [ accept-params ] )  
  
media-range = ( "*/*"  
              | ( type "/" "*" )  
              | ( type "/" subtype )  
              ) *( ";" parameter )  
accept-params = ";" "q" "=" qvalue *( accept-extension )  
accept-extension = ";" token [ "=" ( token | quoted-string ) ]
  
```

Comme souvent, l'ensemble de la syntaxe est expliquée clairement en début de RFC, dans la section *Notational Conventions*, pour ne pas perdre les lecteurs.

Notez que la RFC citée en exemple ci-dessus, celle du format de flux Atom, innove en utilisant des schémas Relax NG pour décrire son balisage.

Vos spécifications phares

En tant que développeur web, voici les principales spécifications qui devraient vous intéresser :

HTML 4.01 (éléments et attributs autorisés)

- Référence : <http://www.w3.org/TR/html4/>
- Version française : <http://www.la-grange.net/w3c/html4.01/cover.html>

XHTML 1.0 (repose sur HTML 4.01)

- Référence : <http://www.w3.org/TR/xhtml1/>
- Version française : <http://www.la-grange.net/w3c/xhtml1/>

CSS 2.1

- Référence : <http://www.w3.org/TR/CSS21/>
- Version française (2.0 !) : <http://www.yoyodesign.org/doc/w3c/css2/cover.html>

Attention, la version 2.1 a beaucoup modifié la version 2.0...

JavaScript 1.5

- DevMo : <http://developer.mozilla.org/fr/docs/JavaScript>
- ECMA : <http://www.ecma-international.org/publications/standards/Ecma-262.htm>

DOM niveau 2

- Noyau : <http://www.w3.org/TR/DOM-Level-2-Core/>
- HTML : <http://www.w3.org/TR/DOM-Level-2-HTML/>
- Événements : <http://www.w3.org/TR/DOM-Level-2-Events/>
- Style : <http://www.w3.org/TR/DOM-Level-2-Style/>

D

Développer avec son navigateur web

C'est à ses outils qu'on reconnaît le bon artisan... Il est effarant de constater combien, encore aujourd'hui, les développeurs web persistent à sous-utiliser leurs navigateurs dans leurs développements. Le navigateur est relégué au rang de visualiseur, ce qui est aberrant !

Il ne s'agit pas seulement de savoir configurer le cache pour être certain de toujours utiliser la dernière version d'une ressource qui change fréquemment. Les principaux navigateurs savent explorer les méandres internes de la page, son DOM, ses CSS, son accessibilité... Ils fournissent des pelletées d'outils informatifs, d'analyseurs, et même des débogueurs JavaScript !

La question délicate du débogage JavaScript fait l'objet d'un chapitre dédié : le chapitre 5. Pour tout le reste (DOM, CSS, etc.), il y a cette annexe.

Il ne tient qu'à vous d'ajouter une bonne dose de confort et d'efficacité, en somme, de *productivité*, à votre méthodologie de développement. Si j'étais vous, je lirais attentivement cette annexe *avant tout le reste*.

Le cache peut être votre pire ennemi

Surtout alors que vous suivez les exemples de ce livre. Le cache, c'est très pratique : ça nous évite de passer notre vie à recharger tout le temps des images, feuilles de style, etc. Mais en développement, cela peut poser problème. On change fréquemment la CSS, le JavaScript, les feuilles XSLT...

Chaque navigateur a sa façon bien à lui de gérer ses caches, qu'il s'agisse du disque ou de la mémoire. Certains seront plus réactifs aux changements de ressources accédées directement qu'à ceux affectant les données obtenues par Ajax ; d'autres rechargeront plus volontiers du JavaScript que des CSS ; etc.

Le rafraîchissement strict

La plupart des navigateurs fournissent un mécanisme clavier pour effectuer un **rafraîchissement strict**, c'est-à-dire un rechargement effectif de l'intégralité des ressources utilisées par la page :

- **Mozilla, Camino, Firefox, MSIE** : au lieu de presser simplement *F5* ou de cliquer sur le bouton idoine, maintenez *Ctrl* ou *Cmd* enfoncée pendant ce temps. Hors MSIE, vous pouvez aussi utiliser *Maj* avec le bouton de recharge ou *F5*.
- **Safari** : utilisez *Cmd+Maj+R* au lieu de *Cmd+R*, ou maintenez *Cmd* enfoncée en pressant le bouton de rafraîchissement.
- **Opera** : il n'y a pas de mécanisme de rafraîchissement strict ! C'est ahurissant, mais c'est comme ça. En théorie, Opera ignore le cache à chaque rafraîchissement explicite avec *F5* ou son bouton de recharge. Dans la pratique, sur les exemples de ce livre, j'ai dû configurer le cache. On peut donc décider de faire de même, pour qu'il vérifie soigneusement les nouvelles versions des ressources (voir un peu plus bas), ou vider complètement le cache (voir ci-dessous).
- **Konqueror** : ignore le cache à chaque recharge explicite avec *F5* ou son bouton de recharge.

Vider le cache

Il est parfois nécessaire de vider complètement le cache, ou en tout cas les parties du cache associées à la page en cours. Voici comment faire.

- **Firefox** : allez dans les options (*Outils > Options* sous Windows, *Édition > Préférences* sous Linux, *Firefox > Préférences* sous Mac OS X) et choisissez la catégorie *Vie privée* puis l'onglet *Cache*. Cliquez sur le bouton *Vider le cache*. Depuis sa version 1.5, Firefox permet également de supprimer instantanément tout ou partie de votre état « privé » en enfonceant *Ctrl+Maj+Suppr* (*Cmd+Maj+Ret.Arr.* Sur Mac OS X) ou

en allant dans *Outils > Effacer mes traces*. Attention, par défaut cela sélectionne bien plus que le cache ! Vous pouvez le configurer depuis la catégorie *Vie privée* des options, avec le bouton *Paramètres*.

- **Mozilla** : allez dans *Édition > Préférences > Cache* et choisissez *Vider le cache*.
- **MSIE 6** : allez dans *Outils > Options Internet > Général > Fichiers Internet temporaires*. Le bouton *Supprimer les fichiers...* permet de vider le cache (pensez à cocher la case *Supprimer tout le contenu hors-ligne*). Validez.
- **MSIE 7 et 8** : allez dans *Outils > Options Internet > Général* et dans la section *Historique de navigation* activez le bouton *Supprimer...* Cliquez alors sur le bouton *Supprimer les fichiers...* et validez.
- **Safari** : dans le menu *Safari*, choisissez *Vider le cache...*, ou pressez *Cmd+Option+E*.
- **Opera** : allez dans *Outils > Préférences > Avancé > Historique* et choisissez le bouton *Vider maintenant*.
- **Konqueror** : allez dans *Configuration > Configurer Konqueror*. Dans la liste de gauche, faites défiler pour choisir la catégorie *Cache*. Cliquez sur le bouton *Vider le cache*.

Configurer le cache

Pour éviter de trop peiner avec le cache, il est important de le configurer correctement. En développement, le mieux est de lui demander de vérifier à chaque requête (à l'aide d'une requête HTTP HEAD), pour chaque ressource, si cette dernière a changé depuis sa dernière version. On peut parfois le désactiver complètement, mais c'est moins utile.

Voici les modes opératoires :

- **Mozilla, Camino, Firefox** : c'est une mauvaise idée, mais vous pouvez demander au navigateur d'allouer 0 Ko au cache dans les options. Toutefois, il ne s'agit que du cache disque : un cache mémoire existe tout de même. Il est désactivable via les options « expert » de `about:config`. Là aussi, mauvaise idée...
- **MSIE 6** : allez dans *Outils > Options Internet > Général > Fichiers Internet temporaires*. Cliquez sur *Paramètres...* Choisissez l'option *À chaque visite de la page*. Ainsi, MSIE vérifiera à chaque fois, et récupérera toute nouvelle version.
- **MSIE 7 et 8** : allez dans *Outils > Options Internet > Général* et dans la section *Historique de navigation* activez le bouton *Paramètres*. Choisissez l'option *À chaque visite de la page*, comme sur les versions 6 et 7.
- **Safari** : à n'utiliser qu'à vos risques et périls... Fermez totalement Safari, ouvrez un terminal (outil *Terminal* dans *Applications > Utilitaires*), et tapez deux lignes : d'abord `rm -fr ~/Library/Caches/Safari`, puis `touch ~/Library/Caches/Safari`. Quittez le terminal.

- **Opera** : allez dans *Outils > Préférences > Avancé > Historique* et configurez les listes déroulantes *Vérifier les documents* et *Vérifier les images*, en les définissant à *Toujours*. Vous pouvez aussi choisir de désactiver le cache en définissant *Cache mémoire* et *Cache disque* à *Off*, mais ce n'est pas une très bonne idée...
- **Konqueror** : allez dans *Configuration > Configurer Konqueror*. Dans la liste de gauche, faites défiler pour choisir la catégorie *Cache*. Si la case *Utiliser le cache* est cochée, assurez-vous que l'option *Assurer la synchronisation du cache* est sélectionnée. Pour désactiver complètement le cache, décochez simplement la case. Mais sur Konqueror, c'est *vraiment* une mauvaise idée : sa gestion du cache est bien adaptée au développement.

Firefox, favori du développeur grâce aux extensions

D'entrée de jeu, Firefox fournit une console JavaScript plutôt correcte (*Outils > Console JavaScript*), mais pas folichonne. Nous l'avons vue au chapitre 5.

Firefox est sans doute néanmoins le meilleur navigateur pour développer des applications web, tant l'univers de ses extensions est riche d'outils fabuleux ! Je n'en cite que deux qui me semblent incontournables : la **Web Developer Toolbar** de Chris Pederick, et **Firebug** de Joe Hewitt.

Firebug

Nous avons déjà vu les possibilités JavaScript de Firebug en détail au chapitre 5. Petite extension légère, il fournit néanmoins une console JavaScript efficace, un objet console utilisable dans vos scripts, un pisteur de requêtes Ajax et un débogueur JavaScript amplement suffisant. Il nous reste à examiner un superbe inspecteur multivue :

- code source, très utile ;
- DOM, plus classique ;
- layout, parfois irremplaçable...

Ces inspecteurs sont de petits bijoux.

L'inspecteur de code source de Firebug est accessible via l'onglet *HTML* de la zone principale. Le code source qui apparaît représente en réalité le DOM de la page ; il est mis à jour en temps réel ! On peut par ailleurs déplier/replier chaque élément. Sélectionner un élément à gauche permet par ailleurs de visualiser à droite le détail des styles qui lui sont appliqués, regroupés par règle CSS, et mettant en évidence les

propriétés qui ont été « écrasées » par d'autres au fil de la « cascade » (les propriétés ainsi ignorées apparaissent barrées).

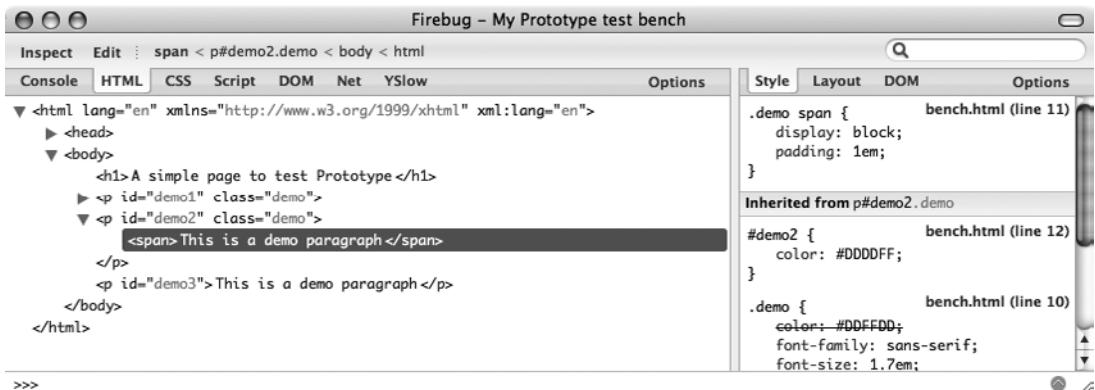


Figure D-1 L'inspecteur de code source de Firebug, en mode « style »

Lorsqu'on survole un élément dans le listing, celui-ci est mis en évidence dans la page elle-même, ainsi que ses marges (en jaune) et ses espacements internes (*padding*, en violet). C'est extrêmement utile !

Sur la droite, outre l'onglet *Style*, on dispose aussi d'un onglet *Layout* qui se révèle indispensable dans bien des cas : il indique la position et l'ensemble des dimensions de l'élément sélectionné dans l'onglet *HTML*, en pixels. Cela permet de vérifier le bon résultat des règles de positionnement qu'on a mises en place, et en cas de doute, un tour par l'onglet *Style* pour vérifier les propriétés *margin*, *padding*, *width*, *height*, *left*, *top*, etc. appliquées permet de découvrir l'éventuel problème.

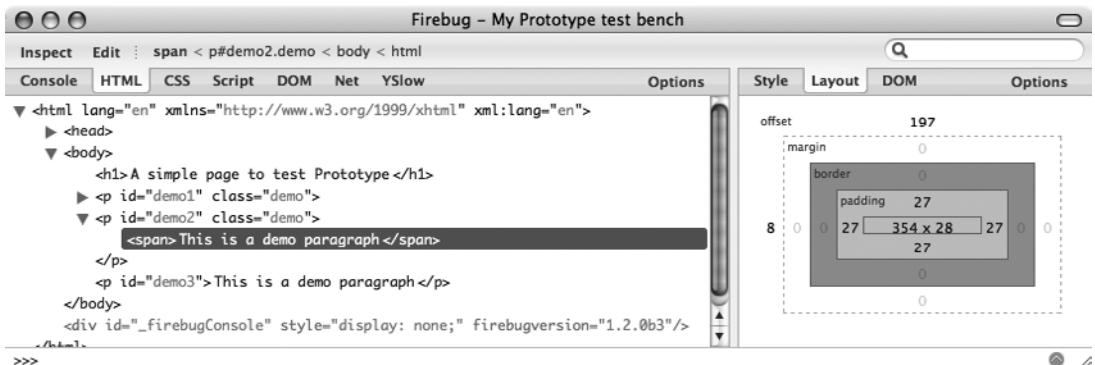


Figure D-2 L'onglet Layout de Firebug se révèle vite incontournable !

Enfin, Firebug propose deux façons d'accéder au DOM :

- À partir de la racine (l'objet `window`), donc des variables dites « globales » ; on y retrouvera par conséquent tous les objets mis en place par les bibliothèques JavaScript chargées (par exemple Prototype), plus les objets « standards » (`window`, `navigator`, `screen`...), mais aussi tous les objets racine plus spécifiques au navigateur (par exemple, sur Firefox, on trouvera `crypto`, `localStorage`...). Ce mode correspond à l'onglet *DOM* de la partie gauche, et n'est que rarement utilisé.
- Pour ce qui concerne l'élément sélectionné dans l'onglet *HTML*, ce qui est évidemment beaucoup plus utile, on y retrouvera les propriétés et méthodes de l'élément, chaque propriété pouvant être « déroulée » si c'est un objet (on liste alors le contenu de cet objet, et ainsi de suite sans limite de profondeur). On y accède par l'onglet *DOM* de la partie droite, et c'est le mode le plus employé.

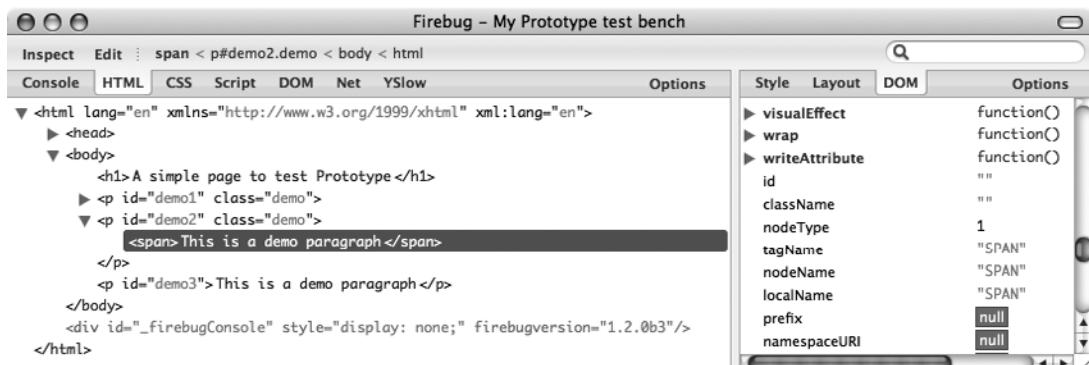


Figure D-3 L'inspecteur DOM pour l'élément sélectionné dans Firebug

Un dernier conseil : pensez à explorer les menus *Options* qui figurent généralement à droite des barres d'onglets. Il y en a pour de nombreux onglets, avec des options dédiées à chaque fois. On y trouve beaucoup de réglages utiles suivant la circonstance...

Web Developer Toolbar

La Web Developer Toolbar de Chris Pederick est devenue une véritable légende. Une fois installée grâce au système d'extensions de Firefox (qui s'est énormément amélioré avec Firefox 3), on obtient la barre d'outils suivante (en français si on l'a téléchargée chez JoliClic¹) :

1. <http://joliclic.free.fr.mozilla/webdeveloper/>

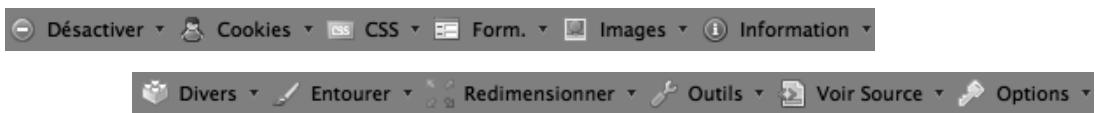


Figure D-4 La Web Developer Toolbar avec icônes et libellés

Parmi la pléthore de fonctionnalités qu'elle propose, je recommande notamment celles-ci, dont je me sers pratiquement au quotidien :

- *Désactiver > Désactiver le cache.* Indispensable lorsque vous souhaitez être absolument sûr(e) que les prochains rafraîchissements, ou les requêtes Ajax, iront directement à la couche serveur.
- *Désactiver > Désactiver JavaScript > Tout le JavaScript.* Très utile lorsque vous suivez les préceptes d'amélioration progressive pour le développement de vos pages, et devez donc vérifier qu'elles fonctionnent convenablement sans JavaScript. N'oubliez pas de rafraîchir ensuite la page pour garantir qu'aucun code JavaScript n'aura été exécuté au chargement.
- *Cookies > Désactiver les cookies > Tous les cookies.* Utile pour vérifier le comportement des sessions et attributions de cookies par le serveur. Ce menu abrite aussi plusieurs options pour nettoyer les cookies par type (session, domaine, chemin...) et consulter/modifier la liste des cookies actifs pour la page.
- *CSS > Désactiver les styles css > Tous les styles.* Une page balisée sémantiquement doit être « lisible » intuitivement sans aucune feuille de style. Ce mode, accessible avec *Ctrl+Maj+S* sur Windows et Linux, *Cmd+Maj+S* sur Mac OS X, se révèle vite indispensable pour « auditer » le balisage d'une page, par exemple dans un contexte de référencement naturel.
- *Images > Remplacer les images par l'attribut alt (tout en bas).* Une page accessible, c'est aussi des attributs *alt=* correctement renseignés pour toutes les images non décoratives (images dites « de contenu »). À l'inverse, des images purement décoratives ont en général un attribut *alt=* défini mais vide. Cette fonction permet de se rendre compte tout de suite de la pertinence des attributs *alt=* en place, et de ceux qui manquent manifestement.
- *Information > Afficher les AccessKeys.* Toujours dans un contexte d'accessibilité, on tente de définir au mieux les raccourcis clavier pour des zones critiques de la page (recherche, navigation, contenu principal, plan du site, bascule de langue ou de taille de texte...). Il n'existe aujourd'hui aucun standard formel sur la question mais des conventions un peu éparses (par exemple <http://clagnut.com/blog/193/>, qui documentent plusieurs schémas dont le standard gouvernemental au Royaume-Uni, l'un des plus complets sur le sujet). Certains navigateurs affichent automatiquement les *accesskeys* sur enfacement de la touche commande de raccourci (*Alt* sur Windows,

Alt ou Ctrl sur Linux, Cmd sur Mac OS X), mais Firefox non : cette option est donc bien utile...

- *Information > Afficher les index de tabulation.* Une page accessible, c'est aussi un ordre de tabulation intelligent, parfois différent de l'ordre naturel des éléments dans le code source du document. C'est particulièrement vrai pour un formulaire, mais toute page gagne à avoir un ordre « utile » de tabulation, par exemple avec le contenu principal en premier, la navigation principale en deuxième et la navigation générique en dernier. Je rappelle que l'attribut `tabindex=` est autorisé sur tout élément visuel (notamment les liens), et pas seulement sur les champs de formulaire.
- *Information > Afficher la profondeur des tableaux.* Lorsqu'on audite la qualité de balisage d'une page, par exemple dans une optique de référencement naturel, la profondeur d'imbrication des tableaux éventuellement utilisés pour la mise en page (horreur !) est d'une importance significative. Au-delà de 3 ou 4 niveaux, le « poids » du contenu est tellement minoré qu'il en devient presque insignifiant. À l'heure où des « recettes » fiables et sans balisage superflu permettent de se passer de tableaux pour pratiquement tous les cas de figure (et de gagner ainsi en elasticité), des mises en page trop lourdes en tableaux imbriqués doivent être détectées et remaniées.
- *Information > Plan du document.* Cette commande très pratique permet d'obtenir, dans un onglet à part, la structure des titres du document, basée sur les balises de titre `h1` à `h6`. On peut ainsi voir en un clin d'œil si la hiérarchie de titres de la page est cohérente ou non. La cohérence de cette hiérarchie influe fortement sur la pertinence de l'indexation par les moteurs de recherche et, dans une optique d'accèsibilité, sur la navigation alternative. Voici les deux principales erreurs de cohérence :
 - hiérarchie ne démarrant pas par le niveau `h1` ;
 - niveaux « sautés » (par exemple des `h4` sous des `h1`).
- *Entourer > Entourer les tableaux > Cellules de tableaux.* Afin de déterminer si une page ayant recours aux tableaux pour tout ou partie de sa mise en page en fait un usage raisonnable ou non, cette commande affiche, avec des codes couleur par niveau d'imbrication, l'ensemble des cellules de tableau du document. On voit immédiatement si celles-ci ne sont utilisées que sporadiquement pour implémenter rapidement la disposition principale de la page (utilisation acceptable) ou si les tableaux sont utilisés à foison et à toutes les sauces (pratique obsolète et dommageable pour le contenu).
- *Redimensionner > 1024 × 768* (nécessite une personnalisation). Inexplicablement, Web Developer Toolbar n'a pas de tailles prédéfinies pour cette commande ! Il faut aller dans la commande Modifier les dimensions... pour définir les combinaisons qui nous intéressent. La taille minimale acceptée aujourd'hui est le 1024×768 .

- *Options > Fonctionnalités persistantes.* Hormis la désactivation du JavaScript, du cache et des cookies, l'ensemble des fonctionnalités de la Web Developer Toolbar « disparaissent » au rafraîchissement ou en naviguant sur une autre page. Si vous souhaitez maintenir une même configuration de visualisation tout au long de ces opérations, cochez cette option.

Une fois qu'on est habitué(e) à la barre, on a souvent tendance à retirer les libellés (champ *Afficher la barre d'outils avec...* de la section *Général* des *Options*).

Figure D-5

La Web Developer
Toolbar en mode
« icônes sans libellés »



Il me faudrait un chapitre entier pour tout décrire, allez plutôt regarder son site web, téléchargez-la si ce n'est déjà fait, et émerveillez-vous !

Vous trouverez pour Firefox énormément d'extensions et d'outils complémentaires, qui correspondront peut-être davantage à vos besoins. Je cite rapidement la XML Developer Toolbar, GreaseMonkey, Platypus, Aardvark...

Les URL correspondantes :

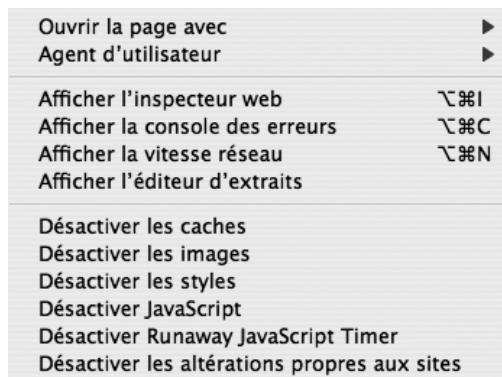
- Web Developer Toolbar : <http://chrispederick.com/work/webdeveloper/>
 - En français : <http://joliclic.free.fr/mozilla/webdeveloper/>
- Firebug : <http://www.joehewitt.com/software/firebug/>
- XML Developer Toolbar : <https://addons.mozilla.org/firefox/2897/>
- GreaseMonkey : <http://greasemonkey.mozdev.org/>
- Platypus : <http://platypus.mozdev.org/>
- Aardvark : <http://karmatics.com/aardvark/>

Les trésors du menu Développement caché dans Safari

Safari 3 dispose d'un menu *Développement*, caché par défaut, qui regorge d'options intéressantes pour le développeur web. Pour l'activer, il suffit de réaliser les manipulations suivantes :

- 1 Fermez totalement Safari.
- 2 Ouvrez un terminal (*Applications > Utilitaires > Terminal*).
- 3 Tapez `defaults write com.apple.Safari IncludeDebugMenu 1`.
- 4 Vous pouvez relancer Safari.

Figure D–6
Le menu Développement
de Safari 3



Dans sa version 2, Safari disposait d'un menu *Debug*, non traduit, qui contenait une pléthore d'options parfois résolument « internes » (du genre *Use ATSU For All Text*, *Use Threaded Image Decoding* ou encore l'incongru *Populate History*). On a désormais quelque chose de plus orienté développeur web. Et par-dessus le marché, ils l'ont traduit !

Parmi les différentes options, la plus utile, et de loin, est *Afficher l'inspecteur web* (*Cmd+Alt+I*). Cet inspecteur, qui peut sembler au premier abord un peu limité, deviendra incontournable en un ou deux clics.

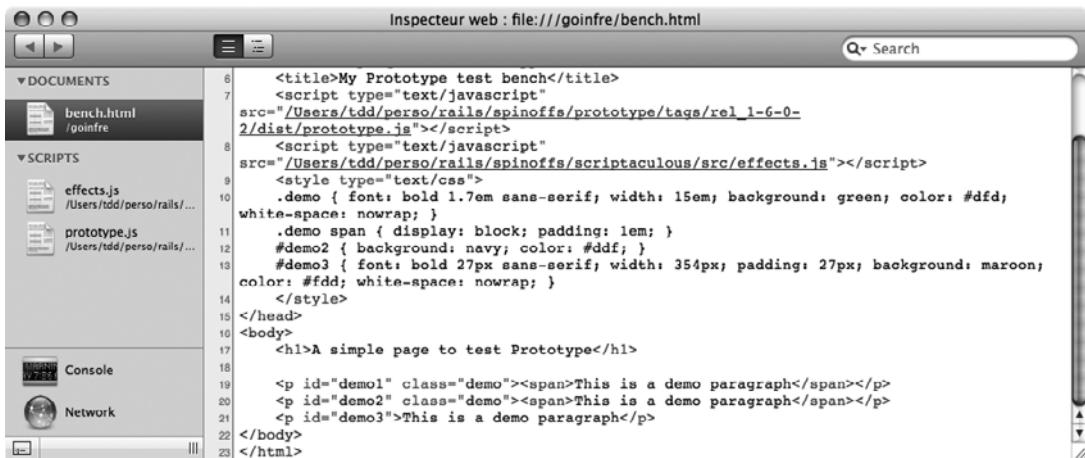


Figure D–7 L'inspecteur web de Safari 3 en mode basique

L'inspecteur web s'affiche par défaut dans une fenêtre à part ; c'est mon mode préféré car en mode « panneau » (le petit bouton en bas à gauche bascule entre les modes fenêtre et panneau), l'inspecteur ne permet pas de redimensionner sa hauteur explicitement, ce qui peut le rendre minuscule...

Autre point important : cet inspecteur peut basculer entre un mode « code source » (celui de la figure D-7) et un mode « arborescence », bien plus utile et similaire à celui de l'onglet *HTML* de Firebug. On bascule de l'un à l'autre à l'aide des deux boutons centraux de la barre supérieure.

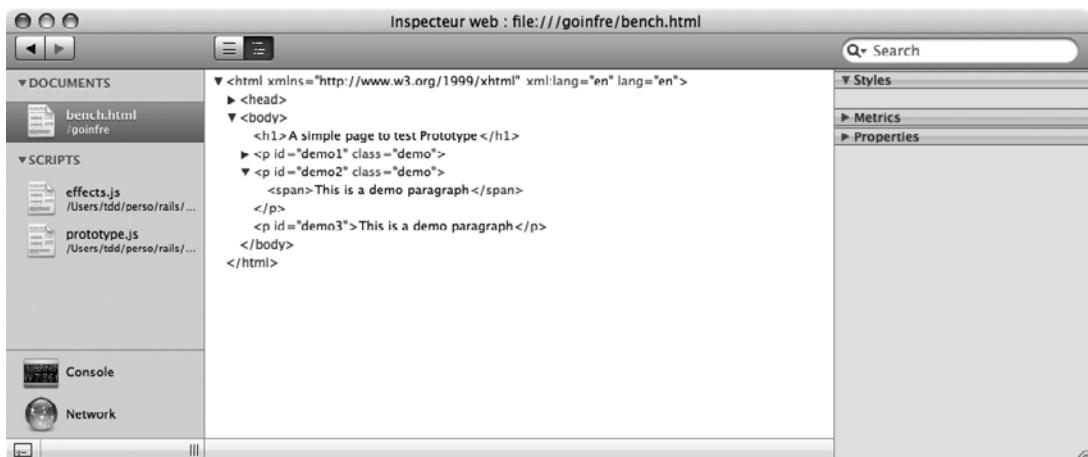


Figure D-8 L'inspecteur web en mode arborescent.

C'est là que se révèle toute la richesse de l'inspecteur : en sélectionnant un élément dans l'arborescence, on dispose des fonctions suivantes :

- Mise en évidence de l'élément dans la fenêtre de navigation par ombrage de tout le reste de la page.
- Affichage des styles actifs dans le panneau supérieur droit
- Affichage des métriques (homologue de l'onglet *Layout* de Firebug) dans le panneau central droit.
- Affichage des propriétés, *regroupées par prototype* (le bonheur !), dans l'onglet inférieur droit. Il faut cliquer sur le prototype qui nous intéresse pour voir les propriétés et méthodes qu'il fournit (pour bien des développeurs qui ne connaissent pas leur *DOM Level 2 Core* et leur *DOM Level 2 HTML* sur le bout des doigts, c'est très instructif !).

Jugez plutôt sur les figures D-9 à D-11 !

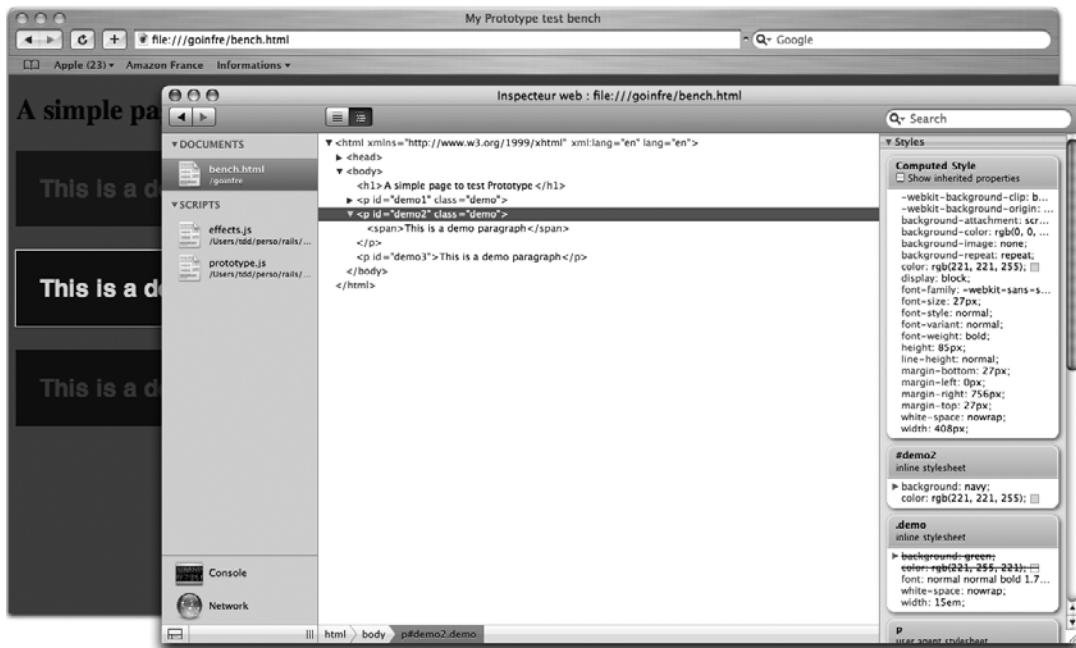


Figure D-9 L'inspecteur web de Safari 3 explore le style d'un élément



Figure D-10 L'inspecteur web de Safari 3 affiche les métriques d'un élément

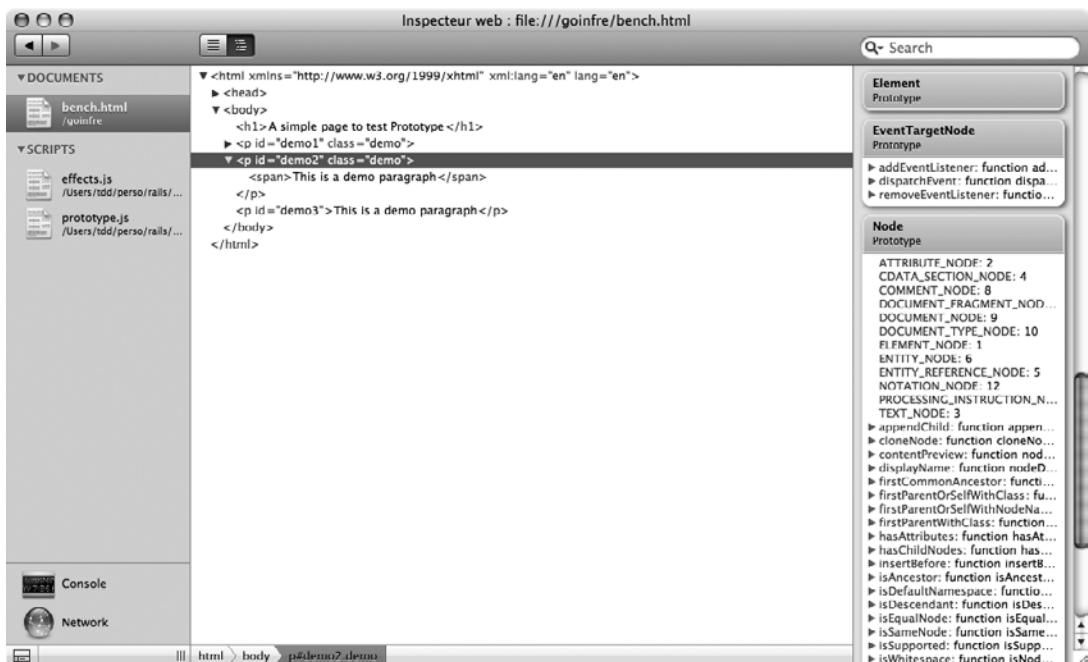


Figure D-11 L'inspecteur web de Safari 3 affiche le DOM d'un élément

Seule ombre au tableau : tout ceci n'est pas modifiable ! On est vraiment en consultation, ce qui ralentit le travail de débogage/prototypage... C'est bien dommage.

Notez également le « chemin » de l'élément dans la page, affiché en bas de l'inspecteur, qui permet de restreindre la vue au contenu d'un élément précis. On trouve le même concept en haut de Firebug dans la vue *HTML*.

Cette fenêtre n'est en réalité par seulement l'inspecteur web, mais l'outil de développement général. Dans le panneau de gauche, les icônes *Console* et *Network* correspondent aux options *Afficher la console des erreurs* (*Cmd+Alt+C*) et *Afficher la vitesse réseau* (*Cmd+Alt+N*) du menu *Développement*.

La « console des erreurs » est en réalité une console JavaScript complète avec possibilité de saisie et d'évaluation immédiate, associée à un objet JavaScript `console` proposant quatre méthodes `error`, `info`, `log` et `warn`, exactement comme dans Firebug. Vos débogages de script à base de `console.log`, par exemple, fonctionnent donc aussi dans Safari 3 ! Attention quand même : l'exécution directe de code JavaScript depuis la console exécute d'abord le code, pour le « loguer » dans la console seulement après coup.

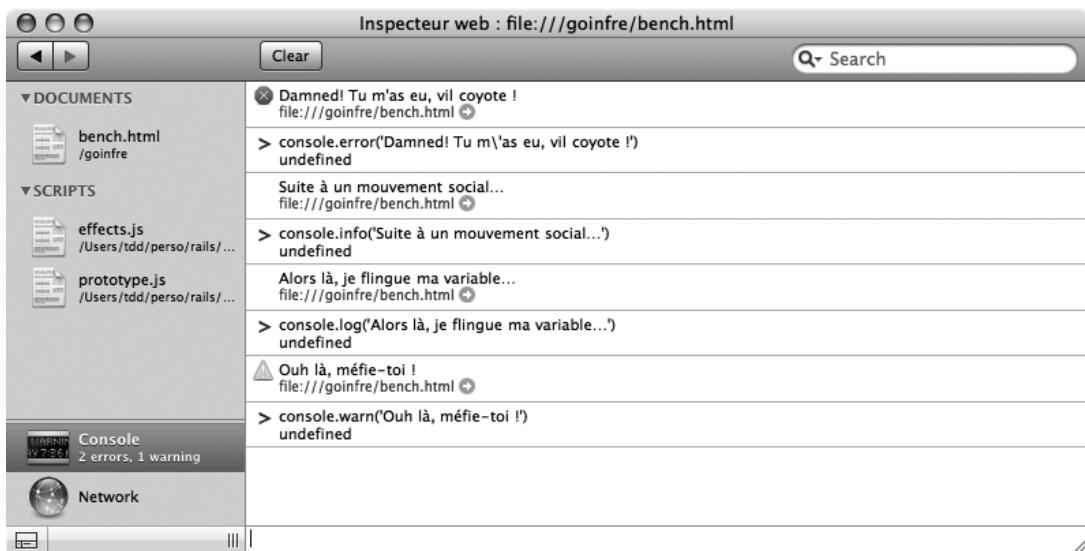


Figure D-12 La console JavaScript de Safari 3 avec des appels à console.

Enfin, l'inspecteur de vitesse réseau affiche le diagramme de chargement des ressources pour la page sur un axe de temps, avec un codage couleur par type de ressource (documents/(X)HTML, images, feuilles de style, autres) et des cumuls. L'axe de temps est calculé automatiquement, sa borne droite correspondant au temps total de chargement de la page.

La figure D-13 montre les résultats de cet inspecteur sur l'accueil du site web de Paris-Web 2008.

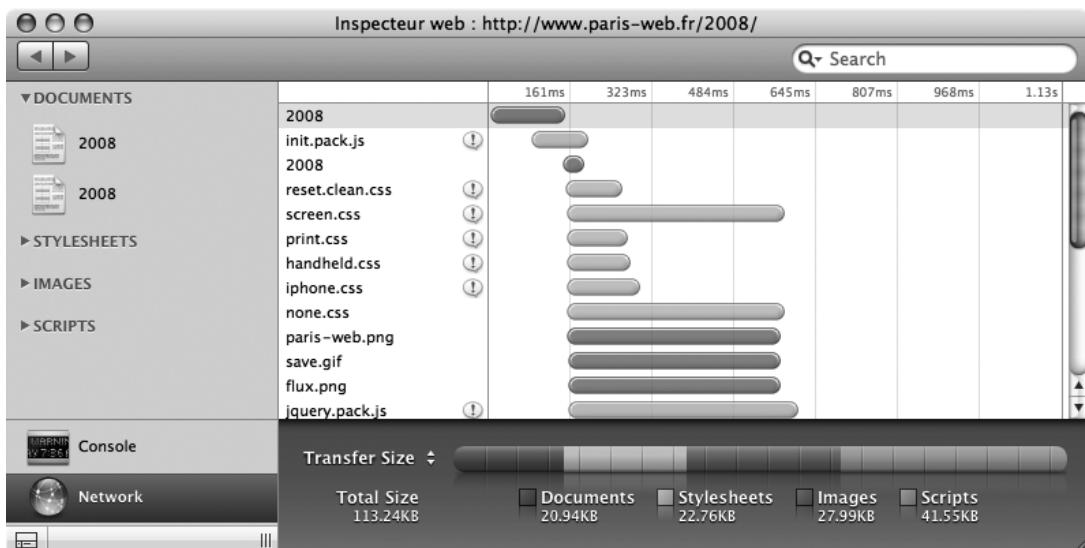


Figure D-13 Inspecteur de vitesse réseau sur Safari 3.

On peut par ailleurs cliquer sur n'importe quelle ligne pour avoir le détail des en-têtes de réponse pour la ressource concernée (et donc examiner l'éventuelle compression par le serveur, la taille ,le type MIME, et j'en passe). Vous en voyez un exemple sur la figure D-14.

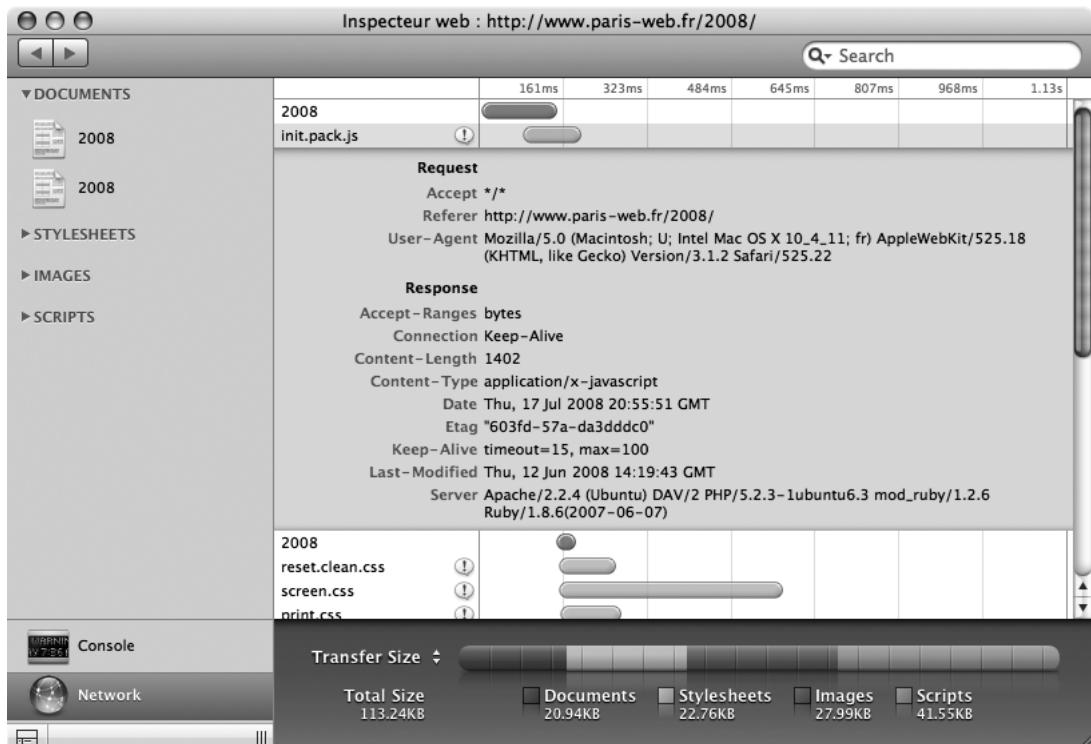
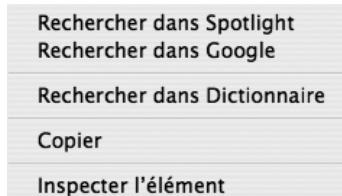


Figure D-14 Inspecteur de vitesse réseau sur Safari 3 avec examen des en-têtes

Pour finir, sachez que l'inspecteur web peut être directement lancé sur n'importe quel élément d'une page : dans Safari 3, quand vous cliquez avec le bouton droit (ou *Ctrl+Clic*) sur un élément, vous disposez d'une option *Inspecter l'élément*, comme on peut le voir en figure D-15. Activer cette option ouvre automatiquement l'inspecteur web sur l'élément concerné.

Figure D-15

Le menu contextuel de Safari 3 permet d'inspecter directement un élément



En somme, Safari 3 n'a pas grand-chose à envier à Firebug (si ce n'est le suivi des requêtes Ajax) !

MSIE 6-7 et la Internet Explorer Developer Toolbar

MSIE ne brille guère par ses possibilités natives. Pas de console JavaScript, une gestion des plus pénibles pour les erreurs et avertissements JavaScript... Toutefois, on peut améliorer la situation avec la **Internet Explorer Developer Toolbar**.

Très fortement inspirée de la Web Developer Toolbar de Chris Pederick (voir pour s'en convaincre <http://www.thinklemon.com/weblog/stuff/WebDeveloperToolbar.jpg>), elle se révèle cependant infiniment moins pratique à l'usage. On la télécharge, comme d'habitude, via une URL ahurissante :

<http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038>

On peut accéder à la barre en question *via* une petite icône supplémentaire en fin de barre d'outils, que vous pouvez voir en figure D-16.

Figure D-16

L'icône de la Internet Explorer Developer Toolbar.



Activer cette icône affiche le panneau correspondant en bas de page, comme on peut le voir en figure D-17.

On retrouve ici une petite partie des fonctions de Firebug et de la Web Developer Toolbar : arborescence du DOM, sélection d'un élément par clic sur la page, mise en avant de l'élément sélectionné, désactivation des images, du cache ou de JavaScript, entourage de certains types d'éléments.

C'est déjà bien, mais l'exécution est particulièrement inefficace.

- On ne peut que *partiellement modifier* le DOM (les attributs seulement, ni ajout, ni retrait, ni modification d'élément).
- L'affichage des styles affiche forcément toutes les propriétés, sans indiquer l'origine de leurs valeurs.
- Une autre vue permet de lister toutes les correspondances de règles CSS pour la page, mais seulement par règle et non par élément, ce qui n'est pratiquement daucune utilité...

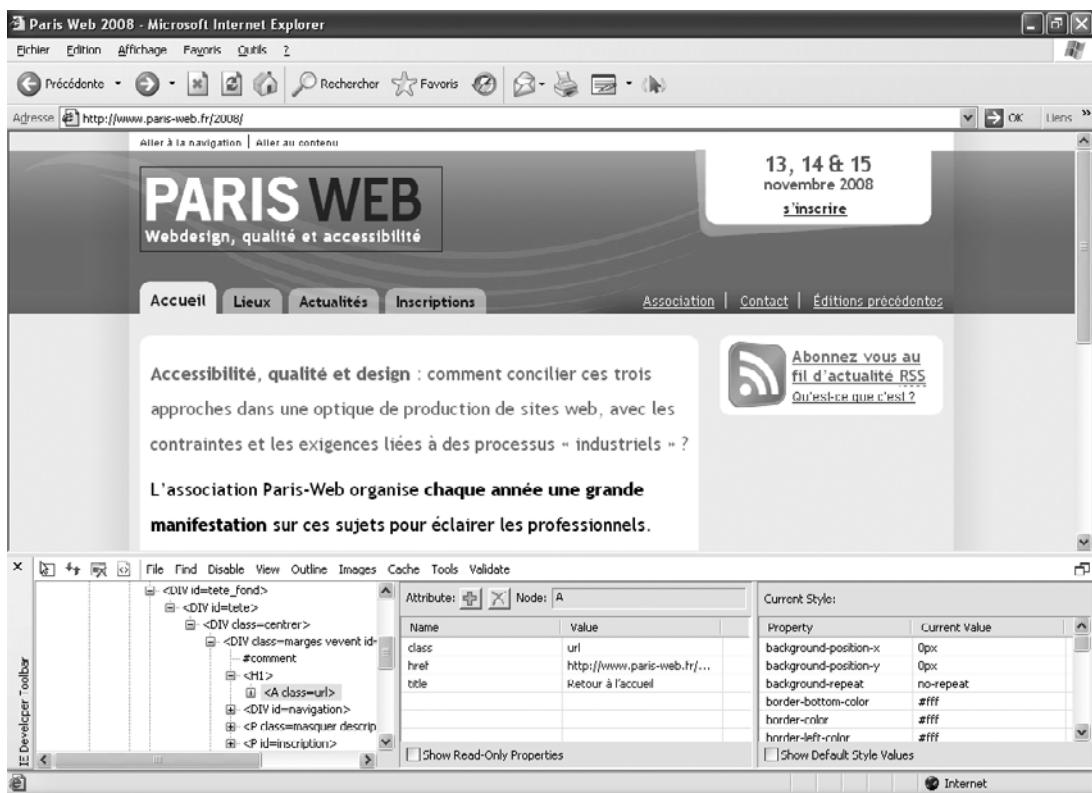


Figure D-17 La IE Developer Toolbar en action

- Aucun débogueur JavaScript n'est présent, on n'a que la console, d'ailleurs très limitée. Il faut recourir à un débogueur externe, soit le Microsoft Script Debugger évoqué plus haut et au chapitre 5, soit des « poids lourds » de la famille Visual Studio (Express ou non), ce qui est démesurément lourd si vous ne les utilisez pas déjà par ailleurs.

Heureusement, la situation s'est un peu améliorée avec la prochaine version 8...

MSIE 8b1 et son outil de développement intégré

La toute dernière version de MSIE a été dotée d'un outil intégré situé, en termes fonctionnels, quelque part entre la Internet Explorer Developer Toolbar, la Web Developer Toolbar de Chris Pederick, et Firebug. Il y a incontestablement des progrès depuis l'ancienne mouture, mais on est encore loin de l'efficacité et de l'ergonomie de Firebug.

On accède à l'outil depuis les menus ou *via* l'icône de barre d'outils qui est identique à celle d'Internet Explorer Developer Toolbar (la petite flèche bleue). On obtient alors le panneau de l'outil en bas de la fenêtre. La figure D-18 montre ce que ça donne en mode « fenêtre séparée ».



Figure D-18 L'outil de développement intégré à IE8

À gauche, trois onglets :

- *HTML* est similaire à l'onglet homonyme de Firebug : il représente le DOM de la page sous format « code source », dépliable et repliable.
- *CSS* présente l'ensemble des CSS de la page, un format trop global pour être utile, un peu comme c'était le cas dans MSIE 6 et 7.
- *Script* est le débogueur JavaScript léger que nous avons déjà étudié en détail au chapitre 5.

À droite en mode *HTML*, trois autres onglets :

- *Style*, qui liste l'ensemble des règles appliquées à l'élément sélectionné dans l'onglet *HTML*. Ce n'est à mon sens pas le bon cheminement dans la consultation (des règles vers l'élément), ça rappelle un peu l'ancienne mouture...
- *Layout*, qui correspond exactement à l'onglet homonyme de Firebug et aux métriques de Safari 3, comme on peut le voir dans la figure D-19.
- *Trace Styles* pour terminer, qui fournit une autre manière d'examiner les styles de l'élément sélectionné : par propriété, avec les origines des valeurs (comprendre : les fichiers et règles qui les ont fournies). Hélas, à voir ce que ça donne (figure D-20), on en vient à se demander si les créateurs de l'outil ont jamais eu à déboguer leurs CSS ! Ce n'est guère utile : pas d'information claire sur quelle règle a « gagné » par rapport à quelle autre, pas d'aperçu des couleurs ni des images... On est bien loin de l'efficacité immédiate de l'onglet *Style* de Firebug. Espérons seulement que les choses iront mieux d'ici la sortie officielle.

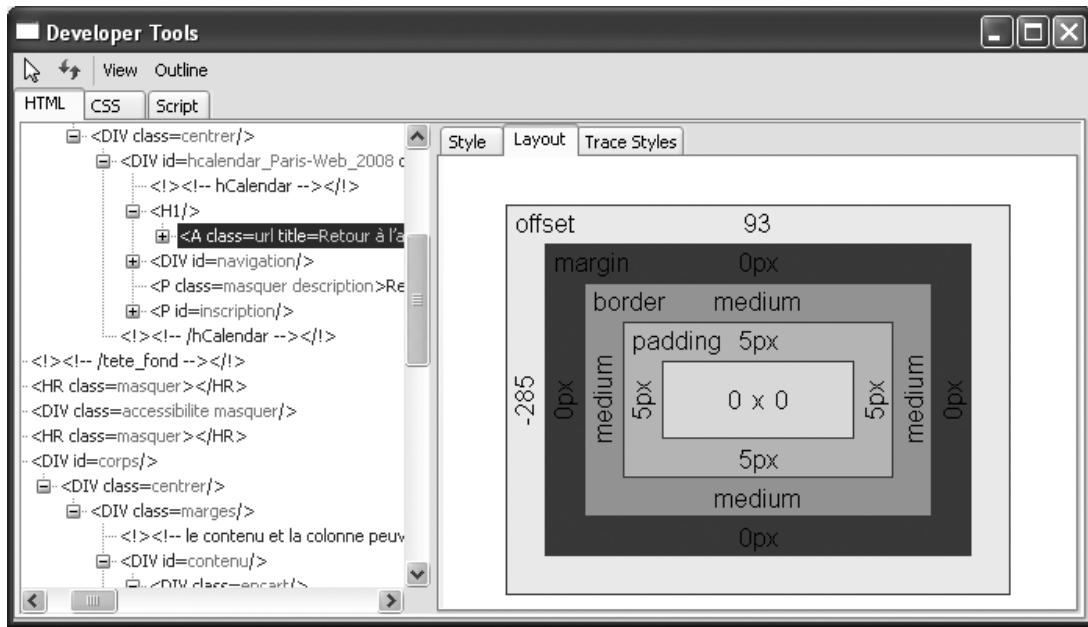


Figure D–19 L'outil de développement intégré à IE8 et son onglet Layout

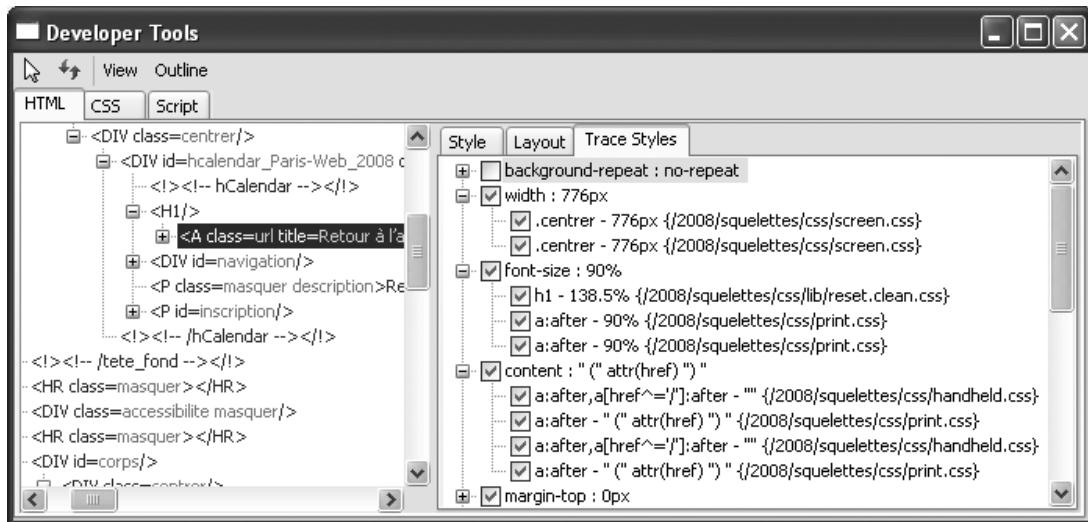


Figure D–20 L'outil de développement intégré à IE8 et son onglet Trace Styles

Quelques précisions complémentaires :

- Dans l'onglet *HTML*, les noms et valeurs d'attributs sont désormais modifiables à la volée, plutôt que de devoir recourir à un encombrant panneau central comme

auparavant. En revanche « c'est œil pour œil, dent pour dent » : on a perdu la possibilité d'ajouter ou supprimer des attributs. Et on ne peut toujours pas ajouter, supprimer ou renommer des éléments.

- La sélection d'un élément dans l'arborescence l'entoure dans la page web correspondante pour faciliter son identification (comme pour les autres navigateurs, la palme revenant à Safari 3 qui met tout le reste de la page dans la pénombre...).
- Le menu *View* de l'outil permet de basculer entre les trois modes de rendu IE :
 - L'ancien mode 100 % propriétaire, baptisé *Quirksmode*, du temps de MSIE 5.x et MSIE 6/7 sans DOCTYPE.
 - Le mode « Strict » introduit par MSIE 6 lorsque la page déclare un DOCTYPE, et qui respecte mieux le modèle de boîte du W3C.
 - Le mode « Standard » de MSIE 8, beaucoup plus respectueux notamment du standard CSS. C'est le mode par défaut de ce navigateur !
- Enfin, le menu *Outline* permet d'entourer rapidement sur la page certains types d'éléments (blocs, cellules de tableaux, éléments positionnés, etc.).

Opera et ses nouveaux outils de développement

Opera est très, très riche de fonctionnalités, notamment en ce qui concerne l'accessibilité. Mais côté développement, il n'offrait pas grand-chose jusqu'à récemment. La firme nordique a fini par monter une équipe dédiée à la réalisation d'outils intégrés s'adressant aux développeurs web, et à force d'être à l'écoute de leurs besoins, a enfanté un rejeton prometteur, baptisé « Dragonfly ».

On y accède par *Outils > Avancé > Outils du développeur (Ctrl+Alt+I)*. Particularité potentiellement ennuyeuse toutefois : l'outil en question est accessible uniquement en ligne. Inutile donc d'espérer pouvoir déboguer un éventuel souci sur Opera lorsque votre ordinateur ne dispose pas d'une connexion réseau rapide (donc, pour le moment, dans l'écrasante majorité des trains et avions, par exemple).

Ceci dit, il faut garder un peu de perspective : la nécessité de déboguer une page (que ce soit sur XHTML, CSS ou le DOM et JavaScript) sur Opera alors qu'elle « passe » impeccablement sur Firefox et Safari est *extrêmement rare*. C'est là tout le bénéfice d'un bon respect des standards ! On peut sans problème concevoir que ces rares occasions auront lieu au bureau, avec une connectivité efficace...

La figure D-21 montre l'outil dans son état initial, une fois déporté dans une fenêtre séparée (ce qui est préférable vu sa disposition interne). On a plusieurs onglets principaux. Nous avons vu l'onglet *Script*, ouvert par défaut, dans le chapitre 5.

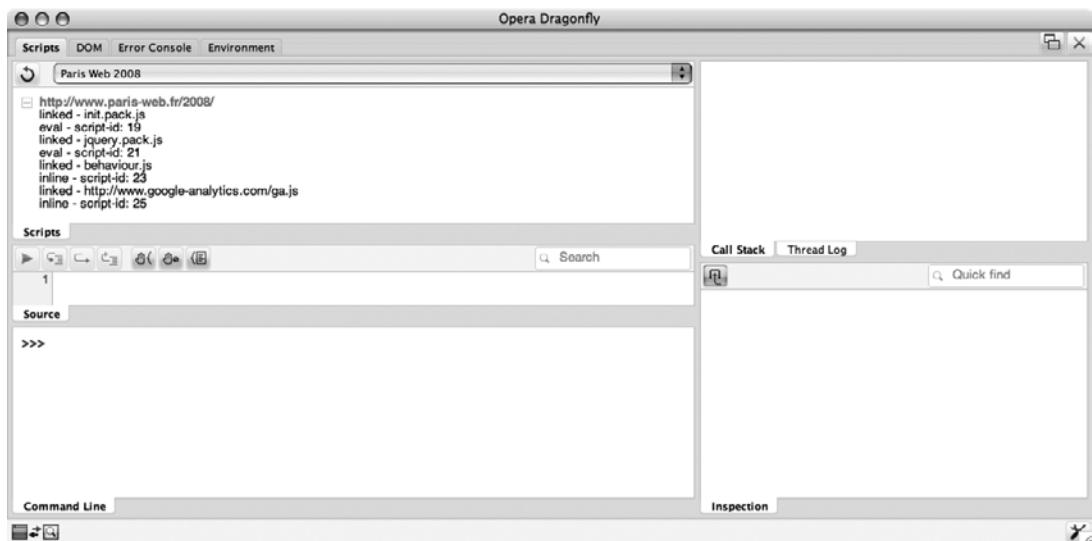


Figure D-21 Opera Dragonfly et sa vue par défaut : le débogueur JavaScript

L'onglet *DOM*, visible plus en détail sur la figure D-22, permet de visualiser d'un seul coup d'œil l'arborescence du document et les propriétés CSS pour l'élément sélectionné (en prenant soin de distinguer celles qui ont été écrasées par d'autres et de les regrouper par feuille de style, façon Firebug).

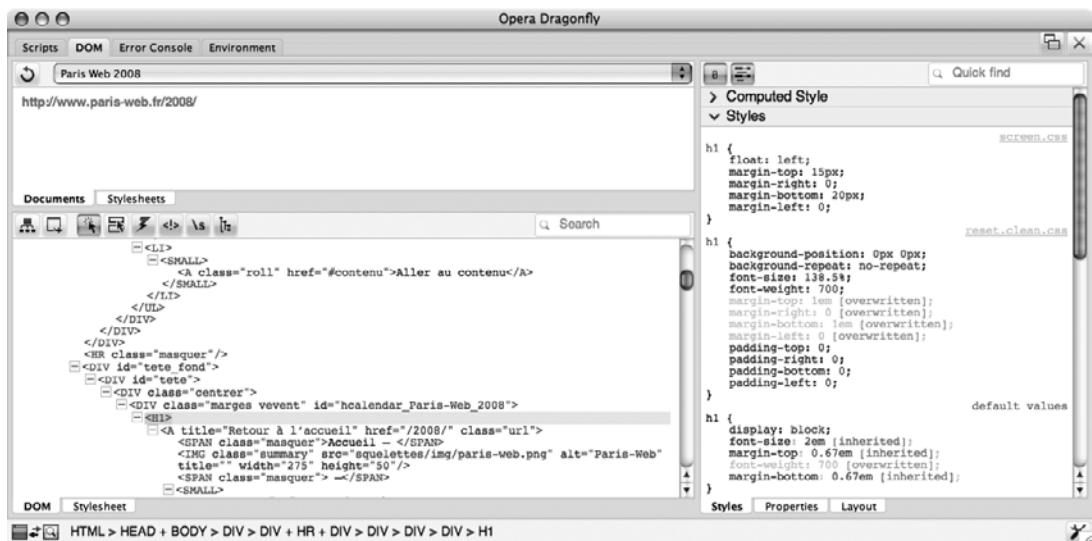


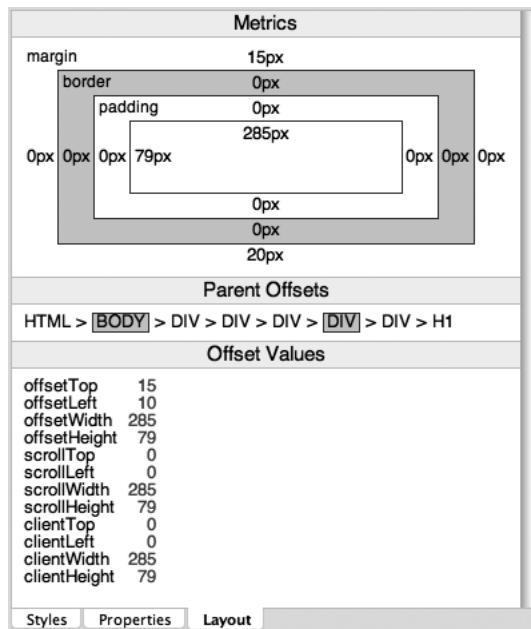
Figure D-22 La vue DOM d'Opera Dragonfly : un parfum de Firebug...

Quelques remarques importantes :

- La vue DOM peut afficher ou masquer les commentaires et espacements superflus ; en revanche, elle n'est qu'en lecture seule.
- Par défaut, lorsque cette vue est active, cliquer dans le document lui-même affiche l'élément cliqué dans la vue DOM, mais ce comportement est désactivable, afin qu'on puisse continuer d'utiliser la page en même temps qu'on examine le DOM.
- Il y a en fait trois onglets à droite : les styles bien sûr, mais aussi *Properties* (les propriétés DOM de l'élément) et *Layout*, montré en figure D-23. C'est exactement l'organisation de la vue HTML de Firebug, mais la vue *Layout* est encore plus détaillée, car elle affiche toutes les propriétés de positionnement, et pas seulement celles du DOM...

Figure D-23

L'onglet Layout de la vue DOM d'Opera Dragonfly : tout sous la main !



- La vue *Error Console* est la console d'erreurs classique. Je trouve toutefois la console d'erreur générale d'Opera beaucoup plus puissante (*Outils > Avancé > Console d'erreur*).
- La vue *Environment* n'est utile que lorsqu'on signale un bogue ou qu'on cherche de l'aide sur les forums : elle donne l'ensemble des informations sur les versions d'Opera et de Dragonfly qu'on utilise.

Avec Dragonfly, Opera a tapé très fort ! Et les habitués de Firebug trouveront immédiatement leurs marques dans l'onglet *DOM*...

E

Tour d'horizon des autres frameworks JavaScript

Lorsqu'il s'agit de bibliothèques JavaScript, ce livre se concentre sur les deux les plus populaires, Prototype et script.aculo.us. Toutefois, il en existe bien d'autres plus ou moins populaires, aux mérites variés. Personnellement, si je devais ponctuellement délaisser Prototype, ce serait sans doute pour YUI, si j'avais des besoins du genre « poids lourd »... En tous les cas, cette annexe vise à faire avec vous un rapide tour d'horizon des principales bibliothèques du marché.

jQuery

Cette petite bibliothèque modulaire, dotée d'un bon système d'extensions, est développée par l'excellent John Resig, qui travaille pour Mozilla. Contrairement à bon nombre des bibliothèques qui suivent, elle se place exactement sur le créneau de Prototype : quelque part entre le simple lissage des incompatibilités JavaScript/DOM et le mastodonte façon SDK...

Ses performances sont bonnes, et elle bénéficie d'une forte croissance de popularité due en bonne partie à son excellent « éco-système », qui se développe au travers de son site officiel et de nombreux sites et forums communautaires, plus quelques livres.

jQuery se concentre sur la concision de la syntaxe et la facilité d'apprentissage. Je trouve pour ma part qu'en dépit d'avantages certains, elle manque de cohérence syntaxique d'une fonctionnalité à l'autre, ce qui rend l'API moins « découvertable » : ce n'est pas parce que tel groupe de fonctions est structuré de telle façon que tel autre suivra les mêmes principes de conception...

J'ajouterais pour finir qu'on a souvent dit que jQuery et Prototype étaient des rivaux acharnés. C'est faux. Les relations entre les concepteurs des deux bibliothèques sont des plus cordiales, au point que John a fait intégrer les suites de tests de Prototype dans le cycle de compilation de Firefox !

Quelques ressources à connaître :

- Le site officiel avec sa documentation, ses démonstrations, ses modules complémentaires et des liens vers les listes de discussions officielles et canaux IRC : <http://jquery.com/>
- Une communauté francophone assez active : <http://www.jquery.info/>
- Bear Bibault, grand fan de bibliothèques JavaScript, a écrit *jQuery in Action* chez Manning : <http://www.manning.com/bibeault/>
- De leur côté, Karl Swedberg et Jonathan Chaffer ont sorti l'excellent *Learning jQuery* : <http://www.packtpub.com/jQuery/book/mid/1004077zztq0>.

Eh oui, il n'existe pour le moment aucun livre sur jQuery en français !

YUI (*Yahoo! User Interface*)

Là aussi, on a assisté cette année à une modification radicale du paysage : il n'y a qu'à voir comment YUI s'est taillé une belle place dans le panthéon des bibliothèques JavaScript.

La grosse différence entre YUI et les autres, c'est qu'il s'agit d'une bibliothèque « corporate », entièrement développée et financée par *Yahoo!*. Rien à voir avec le monde du logiciel libre : seul *Yahoo!* peut la développer. En revanche, elle répond aux besoins très concrets de *Yahoo!* pour ses nombreux sites web et services en ligne ; on y trouve donc un grand soin de la performance, de la modularité, et bien sûr une portabilité à toute épreuve.

C'est en effet YUI qui est au cœur des nombreuses applications *Yahoo!* comme Flickr, del.icio.us ou Upcoming.

YUI est constituée de nombreux modules, qui peuvent être chargés individuellement et dynamiquement grâce à un module de base spécifique. On est là dans la catégorie des poids lourds, avec un noyau dur, des modules multiples allant des effets visuels à

la gestion de sources de données Ajax en passant par la manipulation de l'historique, des cookies, de JSON, et de nombreux *widgets* (calendrier, sélecteur de couleur, menus, éditeur de texte formaté, onglets...). Et j'en passe !

Évidemment, vu les ressources considérables allouées à YUI, on trouve aussi une documentation de tout premier choix et la possibilité de récupérer les modules sur le réseau de distribution rapide (*Content Delivery Network*, ou CDN) de *Yahoo!*.

C'est vraiment un excellent système, surtout si l'on a des besoins extrêmement étendus. Je lui préfère la légèreté de Prototype et j'aime réaliser moi-même mes éléments visuels, mais j'imagine sans peine que pour beaucoup de développeurs web, YUI constitue une solution plus qu'alléchante.

Quelques ressources :

- Le site officiel, un vrai portail : <http://developer.yahoo.com/yui/>
- Un accès direct aux diverses ressources communautaires, notamment les groupes de support : <http://developer.yahoo.com/community/>
- Packt Publishing, qui a sorti *Learning jQuery*, a également permis à Dan Wellman de faire *Learning the Yahoo! User Interface Library* : <http://www.amazon.com/Learning-Yahoo-User-Interface-library/dp/1847192327/>

Dōjō

Attention, mammouth (quoique raisonnablement petit une fois compacté puis compressé) ! Dōjō a démarré dans son coin avec son fondateur Alex Russel puis les gens de SitePen, notamment Dylan Schiemann. Depuis, Dōjō a fait un sacré chemin : la Dōjō Foundation comprend SitePen, bien sûr, mais aussi AOL, IBM et Sun Microsystems, pour ne citer qu'eux ! Gros support *corporate*, donc.

Dōjō est un *framework* complet modulaire, comme YUI. Il se prête particulièrement bien à la réalisation d'applications « desktop-like ». Depuis sa version 0.9, produit d'une refonte à partir de zéro, il se découpe en trois grands modules eux-mêmes très structurés :

- **dojo core**, le noyau. Équivalent à Prototype ou jQuery.
- **digit**, la partie interface utilisateur/*widgets* (système de thèmes visuels, éléments d'interface utilisateur, internationalisation, etc.)
- **dojoX**, qui regroupe les parties expérimentales et les plug-ins.

Le site officiel est superbe, et la documentation de référence est d'excellente facture. On trouve aussi beaucoup de didacticiels en ligne dont le *Book of Dōjō*, assez long et récemment remis à jour.

Dōjō brille particulièrement sur certains aspects innovants comme la gestion des données hors ligne ; Brad Neuberg a présenté il y a plus d'un an déjà Dōjō Offline sur la base de la technologie Google Gears...

Les ressources fondamentales et les bons bouquins :

- Le site officiel, la documentation, les listes de discussion, etc. : <http://dojotoolkit.org/>
- *Mastering Dōjō*, par l'équipe noyau de Craig Riecke, Rawld Gill et Alex Russel : <http://www.pragprog.com/titles/rgddojo/mastering-dojoo>
- *Dōjō: Using the Dōjō JavaScript Library to Build Ajax Applications* par James E. Harmon : <http://www.amazon.com/dp/0132358042/>
- *Dōjō: The Definitive Guide* par Matthew A. Russel : <http://www.amazon.com/dp/0596516487>

Ext JS

Ext JS est en fait un dérivé de la version 1.0 de YUI, porté par Jack Slocum, qui a démarré cette branche en 2006. On lui doit notamment les fondations des moteurs de sélection DOM par CSS qu'on retrouve aujourd'hui dans la plupart des autres frameworks.

Ext JS a toujours fourni toute une série de modules applicatifs, bien écrits, avec un système de thèmes, des grilles de données, des onglets, des menus, des gestionnaires de disposition... Tout à fait dans la cour de Dōjō et, logiquement, de YUI.

L'un des points les plus notables de Ext était qu'il avait été écrit de manière si modulaire et découpée qu'on pouvait le baser sur divers noyaux tiers comme jQuery ou Prototype ! Une performance impressionnante...

La version 3.0, prévue pour l'hiver 2008/2009, devrait inclure de nombreuses innovations et nouveautés comme la prise en charge de Comet/Bayeux (un point fort traditionnel de Dōjō) et une meilleure prise en charge d'ARIA, ce que je ne peux qu'encourager pour cette gamme de frameworks.

Aujourd'hui, Ext a grandi et une société a vu le jour. Elle s'appelle Ext LLC et édite deux variantes du framework : Ext JS qui est 100 % JavaScript, et Ext GWT qui fournit la même fonctionnalité mais en se basant sur GWT, dont l'approche est radicalement différente, comme nous le verrons plus loin. La société propose aussi un support technique commercial. Je ne suis pas contre ce modèle en soi, mais il est dommage que cette bascule commerciale ait, dans la pratique, sonné le glas d'une bonne partie de l'aspect « logiciel libre » du projet, et notamment de certains projets dérivés d'encapsulation GWT...

- Le site officiel est bien fait, les documentations de référence sont de bonne qualité et on trouve pas mal de didacticiels et d'exemples : <http://extjs.com>.
- En revanche, la récupération commerciale semble avoir engendré un léger désintérêt des auteurs face à une communauté (pourtant florissante encore récemment) qui est devenue un peu captive de l'entité commerciale : impossible de trouver le moindre livre sur Ext JS, même en anglais...

Base2

Base2 nous vient de Dean Edwards. Et Dean, tous les *JavaScript ninjas* le révèrent. Développeur britannique en *freelance*, il est un peu comme un ermite qui, occasionnellement, descend de sa montagne et nous balance négligemment une pure petite merveille de JavaScript, qu'on met tous des heures, si ce n'est des jours, à comprendre. Ensuite, on s'en inspire et on sort les frameworks plus connus ; du script de Dean, ça se distille doucement, mais sûrement. Au bout de la cornue, le grand public...

Base2 est une sorte de plus petit dénominateur commun pour les autres frameworks : il se concentre sur le lissage des incompatibilités entre les navigateurs, et la mise au point d'un système d'héritage de classes. Pour tout le reste (Ajax, le glisser-déplacer, les effets, etc.) il laisse les autres s'amuser, ce n'est pas son problème : pour Dean, une fois qu'on a envoyé paître les innombrables petits obstacles de fond, on peut « laisser l'exercice au lecteur »...

Base2 fournit donc une sélection DOM rapide basée sur CSS, un DOM enfin conforme au niveau 2 officiel du W3C, l'événement `DOMContentLoaded` (équivalent du `dom:loaded` de Prototype), quelques méthodes fort utiles d'obtention de style calculé ou de position, et tout ça... à partir de MSIE 5.0 !

D'ailleurs, Dean est tellement fort qu'il est aussi l'auteur de l'immensément célèbre **IE7.js** (qui justement servait à torturer IE5 et IE6 pour qu'ils soient équivalents à IE7 ; il a depuis ajouté des trucs dans **IE8.js**, le malin), du très bon compresseur de script **/packer/**, et du premier système de sélection DOM par CSS, **cssQuery**, sur les ferments duquel tous les autres (y compris Jack Slocum pour Ext JS) ont basé leur travail (au moins au début).

Évidemment, Base2 ne séduit pas tellement les foules car il est trop « bas niveau », au sens où il ne fournit aucun service avancé côté utilisateur. Comme les autres frameworks réimplémentent l'équivalent, les développeurs web ne voient pas trop l'intérêt de Base2. Du coup, pas un bouquin en vue, et la documentation, bien que présente et complète, est succincte et sans exemples.

Mais je ne pouvais pas passer Base2 sous silence. Chaque nouvelle version suscite l'admiration et crée l'inspiration chez les développeurs des *autres* bibliothèques.

Le site officiel : <http://code.google.com/p/base2/>.

MooTools

MooTools est arrivé un peu après la plupart des bibliothèques que j'ai citées ici. Commençons par les bonnes nouvelles : ça marche plutôt bien, la documentation est correcte, et c'est disponible *via* le CDN de Google, tout comme jQuery, Prototype et script.aculo.us.

MooTools est très modulaire et joue beaucoup sur ce point, en proposant deux *builders* en ligne qui permettent de choisir les modules que l'on veut, le type de compression que l'on souhaite (YUI Compressor, JsMin ou rien), et d'obtenir un script unique qui répond à tout ça. Le seul autre framework ayant un système de ce type aussi facile d'emploi, c'est YUI.

Enfin, MooTools, c'est une petite équipe motivée dirigée par le talentueux Valerio Proietti.

Toutefois leur popularité ne décolle pas vraiment. MooTools se classe plutôt dans la catégorie de jQuery et Prototype, et n'a pourtant pas la moitié de leur popularité suivant les sondages ; YUI le devance de loin. À quoi cela est-il dû ?

Je ne peux pas répondre à cette question avec certitude, mais peut-être est-ce dû à ceci (je parle pour moi, et seulement moi) : MooTools n'a, à l'heure où j'écris ces lignes, rien de significatif qu'on ne retrouve pas dans jQuery, Prototype ou YUI, mais surtout, *il est généralement en retard*.

Il y a évidemment des gens talentueux dans l'équipe MooTools (au premier rang desquels Valerio), mais le schéma qu'on ne cesse de constater est le suivant : une nouvelle fonction apparaît dans un des trois frameworks cités plus haut, et dans MooTools quelques jours ou quelques semaines plus tard. Pas forcément du copiage à proprement parler, mais tout de même, il y a eu quelques cas édifiants, qui ont nuis à la réputation du framework et de son équipe.

Ajoutez à cela que certains anciens membres de l'équipe MooTools ont parfois dénigré un peu trop fort les autres frameworks en public, et vous comprendrez pourquoi le framework subit une réputation qui ne lui fait guère de bien. C'est dommage, parce qu'on y trouve du très bon code et que l'équipe a de nombreux membres de valeur, mais c'est ainsi.

- Le site officiel fournit la documentation de référence, les fameux *builders*, le blog de l'équipe et des exemples et démonstrations : <http://mootools.net>
- Je cite aussi un micro-moteur d'effets visuels (3 Ko compressé !) écrit par Valerio, qu'on peut utiliser avec MooTools ou Prototype : **moo.fx²**, téléchargeable sur <http://moofx.mad4milk.net/>.

Et deux autres, très liés à la couche serveur...

Jusqu'à présent, nous avons vu des frameworks 100 % JavaScript, qui sont agnostiques quant à la couche serveur : on peut s'en servir aussi bien sur du PHP que du Ruby on Rails, du Java EE, de l'ASP.NET, du ColdFusion, et même de bons vieux CGI en Perl ou en C !

Il existe cependant deux frameworks assez répandus, ou en tout cas assez connus, qui méritent d'être cités. L'un d'eux est spécifique à une couche serveur ASP.NET, et l'autre à une couche serveur Java.

GWT

Le *Google Web Toolkit* est une API Java extrêmement riche dont le principe fondamental est simple : vous écrivez votre interface web en Java, un peu comme vous décririez une interface en Swing. GWT se charge de compiler tout ça pour pondre du HTML, des CSS et du JavaScript qui passent partout, sont performants et interagissent de façon transparente avec votre couche serveur, en gérant les problématiques de persistance d'état, de flux de données, etc.

Autant je suis le premier à regarder avec avidité toutes les nouvelles API que nous sort Google (j'ai notamment déjà travaillé avec les API Maps, Maplets, Static Maps, Chart, Feedburner, Analytics, Checkout, Custom Search, Gears, OpenSocial, Sitemaps et YouTube), autant je suis plutôt du genre à fuir GWT.

C'est tout à fait personnel : étant moi-même férus de JavaScript avancé et très perfectionniste sur la sémantique et l'accessibilité de mon balisage, l'idée de laisser tout ça à une boîte noire (enfin non, c'est du logiciel libre, mais pour se farcir la masse de code et l'internaliser, bonjour...) qui, pour passer *vraiment* partout, fait forcément bon nombre d'entorses à mes idéaux, m'est plutôt douloureuse.

Il appartient à chacun de se faire sa propre opinion, comme toujours. La promesse de GWT a bien des aspects alléchants pour un chef de projet : des performances potentiellement excellentes, l'intégration avec de nombreux EDI (notamment pour

le débogage), la capitalisation des notions issues de Swing (gestionnaires de disposition, etc.), et de très nombreuses fonctionnalités.

Et surtout, ça évite d'avoir besoin de fortes compétences en JavaScript ; on peut réutiliser nos développeurs Java confortablement encadrés par le typage statique et les génériques, les avertissements du compilateur, la complétion automatique, la refactorisation, et tous les autres garde-fous.

Mais finalement, c'est fonctionnellement presque identique à Dōjō ou YUI... Avec en plus une contrainte forte côté serveur. D'ailleurs, sur les sondages de l'univers Ajax, GWT arrive loin derrière tous les frameworks cités précédemment.

On parle ici d'un projet Google massif et avec de forts investissements *corporate*, donc forcément, un très gros site et d'innombrables documentations : <http://code.google.com/webtoolkit/>.

ASP.NET AJAX (ex-Atlas)

Étant doté d'une éthique de fer, je cite ASP.NET AJAX.

Bon, je vide mon sac tout de suite : je suis contre l'idée même d'un serveur web tournant sur plate-forme Windows (c'est le cas pour l'écrasante majorité des applications .NET), et je refuse l'idée qu'il me faut plusieurs Go de disque pour installer un EDI qui servirait à faire du Web (sans parler du peu de confiance que j'accorde, personnellement, à des gens qui sont à l'origine de Windows, Access, Notepad, Visual Basic et Paint).

Maintenant, je comprends parfaitement l'intérêt que peut présenter pour une équipe de développement le haut niveau d'intégration que propose Visual Studio.NET, ou même sa version Visual Studio Web Express (« juste » 1,6 Go pour l'installation, une paille). Et C# est vraiment très loin au-dessus des autres langages pris en charge par la CLR (ce qui est notamment dû au fait qu'il est piloté par l'immense Anders Hejlsberg, qui nous avait auparavant donné Delphi). Si vous faites déjà principalement du .NET en C# (par pitié, rendez-nous service, laissez tomber VB.NET...), capitaliser là-dessus semble être du simple bon sens.

Un petit rappel quand même : ASP.NET ne sait pas vraiment, sauf erreur de ma part, faire du MVC (on bidouille, mais on n'a pas un vrai découpage), et sa première tentative vers Ajax, à savoir les premières moutures d'Atlas, était tout simplement catastrophique de dysfonctionnements.

Tout le monde a droit à une deuxième chance, et justement le système a été renommé récemment en ASP.NET AJAX (au moins, on sait tout de suite de quoi ça parle), intégré à ASP.NET 3.5 et téléchargeable en complément de la version 2.0 (et pour une fois c'est petit, à peine 1,4 Mo !).

Très honnêtement, je ne sais pas si c'est pratique, utilisable et sans bogue, mais sa popularité semble arriver juste après celle de YUI, avant MooTools. Ça doit donc être plutôt pas mal !

Le site officiel, évidemment truffé de vidéos, didacticiels et documentations variées (l'avantage d'émaner du plus gros éditeur de logiciels au monde) : <http://ajax.asp.net>.

Petit tableau récapitulatif

Voici un petit tableau qui récapitule les différents frameworks purement JavaScript (j'exclus donc GWT et ASP.NET AJAX, qui jouent sur un autre tableau).

Tableau 5-1 Tableau E-1 Comparaison rapide des principaux frameworks

	Prototype	jQuery	YUI	Dōjō	ExtJS	Base2	Mootools
Version	1.6.0.3	1.2.6	2.5.2	1.1.1	2.2	1.0β2	1.2
Taille noyau	140 Ko	98 Ko	31 Ko	260 Ko	160 Ko	42 Ko	90 Ko
Taille noyau compressé	32 Ko	14 Ko	11 Ko	24 Ko	25 Ko	6 Ko	20 Ko
CDN	Google	Google	Yahoo!	Google	-	-	Google
Espace de noms	-	jQuery	YUI	dōjō	-/Ext	plusieurs	-
Licence	MIT	MIT/GPL	BSD	BSD/AFL	commerciales et GPL	MIT	MIT
Taille équipe	10	5	10	moult	5	1	9
Éditeur	Sam Stephenson	John Resig	Yahoo!	Dōjō Foundation	Ext JS, LLC	Dean Edwards	Valerio Proietti

La vraie fausse question de la taille...

Pour terminer, un mot sur la préoccupation inopportunе qu'ont beaucoup de gens au sujet de la *taille* du framework, en termes d'impact sur le temps de téléchargement, et donc d'initialisation, de leurs pages.

Cette préoccupation est totalement infondée. Ne choisissez *en aucun cas* votre framework sur cette base. En effet, vos pages font régulièrement appel à des tas de ressources bien plus lourdes que le framework, et qui varient beaucoup plus souvent. Une petite mise en perspective s'impose...

Sur vos pages, il y a souvent :

- Une image JPEG 16/9 « pleine largeur » sur un site en 940 pixels de large, format extrêmement classique ne serait-ce que pour un fond ou un bandeau d'accueil, va peser dans les 70 Ko, déjà compressée (vers 75 % de qualité JPEG, ce qui est relativement nominal).
- Une animation Flash pour un menu ou une page d'accueil pèse régulièrement au moins 40 Ko, déjà compressée.
- Ces images varient souvent fréquemment, voire d'une page à la suivante, nécessitant un rechargement fréquent.

Et côté JavaScript ?

- Le plus gros framework du tableau précédent, une fois compressé correctement, pèse 32 Ko.
- La plupart de ces frameworks sont disponibles *via* un CDN, qui va donc décharger votre serveur et garantir un téléchargement optimal en vitesse.
- Ces frameworks ne changent que lorsque vous passez à une nouvelle version ; une fois téléchargés une première fois, si vous utilisez un CDN ou avez configuré correctement votre propre serveur, les navigateurs les gardent en cache aussi longtemps que vous le souhaitez (un mois, un an...). Ils sont donc disponibles instantanément, contrairement à vos animations, images et pages web en général.
- Même sur une mauvaise connexion ADSL (1 Mbit/s en descente, soit entre 5 % à 15 % d'une bonne connexion métropolitaine), l'écart maximal de temps de téléchargement entre les versions compressées des frameworks ci-dessus est de l'ordre *d'un dixième de seconde*. Même en version *décompressée*, on ne dépasse pas 1,8 s d'écart. Et ce, sur la première page uniquement.

La conclusion s'impose d'elle-même : oubliez les questions de taille.

Index

Symboles

==, opérateur JavaScript 35

A

accessibilité

 Ajax et l'accessibilité 391

 JavaScript 103

ActiveScript 22

addEventListener 96

AHAH, microformat 285

AJAX

 Ajax.Response 309

 asynchronous (option de requêteur) 306

 contentType (option de requêteur) 306

 encoding (option de requêteur) 307

 evalJS (option de requêteur) 307

 evalJSON (option de requêteur) 307

 evalScripts (option de Ajax.Updater) 311

 Fonctions de rappel 308

 getAllHeaders (méthode de

 Ajax.Response) 309

 getAllResponseHeaders (méthode de

 Ajax.Response) 309

 getHeader (méthode de Ajax.Response) 310

 getHeader (méthode de requêteur) 306

 getResponseHeader (méthode de

 Ajax.Response) 310

 headerJSON (propriété de

 Ajax.Response) 309

 insertion (option de Ajax.Updater) 311

 isSameOrigin (méthode de requêteur) 306

 method (option de requêteur) 306

 onComplete (option de requêteur) 307

 onCreate (option de requêteur) 307

 onException (option de requêteur) 307

 onFailure (option de requêteur) 307

 onInteractive (option de requêteur) 307

 onLoaded (option de requêteur) 307

 onLoading (option de requêteur) 307

 onSuccess (option de requêteur) 307

 onUninitialized (option de requêteur) 307

 onXYZ (option de requêteur) 307

 parameters (option de requêteur) 307

 postBody (option de requêteur) 307

 readyState (propriété de Ajax.Response) 309

 request (propriété de Ajax.Response) 309

 requestHeaders (option de requêteur) 307

 responseJSON (propriété de

 Ajax.Response) 309

 responseText (propriété de

 Ajax.Response) 309

 responseXML (propriété de

 Ajax.Response) 309

 sanitizeJSON (option de requêteur) 308

 status (propriété de Ajax.Response) 309

 statusText (propriété de Ajax.Response) 309

 success (méthode de requêteur) 306

 transport (propriété de Ajax.Response) 309

AJAX (Asynchronous Javascript And XML)

 Ajax et l'accessibilité 391

 anatomie d'une conversation 245

 avoir le bon recul 390

 bien choisir son type de réponse 266

 c'est quoi au juste ? 13

 exemple

 barre de progression (JS) 293

 d'un traitement 275

 nombre d'articles de blogs

 DOM3 XPath 287

 GoogleAJAXSLT 291

A
AJAX (*suite*)

- saisie de commentaires 280
 - Prototype 312
- sauvegarde automatique 267
 - Prototype 304
- suivi de marchés 296

réponse

- en texte simple 267
- JavaScript 293
- JSON 296
- XHTML 280
- XML avec XPath 286
- simplifié avec Prototype 303
- surveiller les échanges 263

Ajax.XSLTCompleter 451**Amazon.fr** 407

- obtenir une clef API 408
- réponse REST 411
- requête REST 409

API REST 406**Archive de codes sources XLII****Atom XXIX**, 487

- entries 488
- exemple
 - consulter le Standblog 499
- feed 488

attachEvent 98**B****balisage sémantique** 556

- avantages insoupçonnés 552
- exemple
 - didacticiel technique 575
 - formulaire sémantique et accessible 566
 - tableau de données à en-têtes groupés 573

binding (JavaScript) 47**Brendan Eich XXX****C****cache**

- configurer le cache 633
- rafraîchissement strict 632
- vider le cache 632
- votre pire ennemi 632

codes sources

- archive XLII
- complétion automatique de texte 1, 373
- contraintes de sécurité sur le navigateur 402
- CORBA (Common Object Request Broker Architecture) 405
- CRUD (Create, Read, Update, Delete) 406
- CSS XXVIII**
 - CSS 2.1 XXXVI
 - CSS (Cascading StyleSheets)
 - cascade 584
 - descriptions de propriétés (spécifications) 615
 - éléments en ligne et de type bloc 591
 - éléments remplacés 591
 - fusion des marges 591
 - groupement 594
 - héritage 586
 - jargon 583
 - marge, bordure et espacement 590
 - modèle des boîtes 587
 - modèle Microsoft 592
 - prise en charge actuelle 582
 - propriétés 583
 - de contenu généré automatiquement 597
 - de couleurs et d'arrière-plan 599
 - de formatage visuel 596
 - de gestion de la police de caractères 599
 - de gestion du corps du texte 602
 - de l'interface utilisateur 603
 - de pagination 598
 - des tableaux 603
 - du modèle des boîtes 596
 - tour d'horizon) 595
 - pseudo-éléments 594
 - règle 583
 - sélecteur 583
 - tour d'horizon 593
 - spécificité (calcul) 585
 - unités absolues et relatives 588
 - valeurs spécifiée, calculée, utilisée, concrète 587
 - versions 582
 - CSS Zen Garden XXXIII

D

Dave Winer XXXI, 486
DCOM (Distributed Component Object Model) 405
del.icio.us XXXVIII
Digg XXXVIII
divitis 7
Document, interface DOM 74
DOM (Document Object Model) XXVIII, 63
 arborescence d'objets 68
 bonnes habitudes 83
 construction avant scripting 85
 descriptions de propriétés et méthodes (spécifications) 616
 déetecter le niveau 83
 document, interface 74
 DOMImplementation, interface 80
 écueils classiques 115
 Element, interface 76
 événement 94
 propagation 99
 exemple
 décoration automatique de labels 106
 validation automatique de formulaires 111
hasFeature 80
HTMLDocument, interface 81
HTMLElement, interface 82
inspecteur 87
modules (ou aspects) 67
MSIE et le DOM de select/option 116
NamedNodeMap, interface 78
niveaux 65
Node, interface 70
NodeList, interface 78
ordre de création des nœuds 84
problèmes résiduels 116
propriétés de déplacement 71
répondre aux événements 94
Text, interface 77
 types de noeuds 71
DOMImplementation, interface DOM 80
Dotclear XXXIX
DTD (Document Type Definition) 555, 607, 618

E

ECMA XXX, 20
 TG1 21
ECMA-262 XXX, 20
EcmaScript 21
Element, interface DOM 76
événement
 accessibilité 103
 annuler le traitement par défaut 103
 bouillonnement 100
 capture 101
 MSIE 115
 objet Event (DOM) 102
 propagation 99
 récupérer l'élément déclencheur 102
 répondre aux 94
 stopper la propagation 102
Event, objet
 DOM 102
 Prototype 197
exemple
 Amazon.fr 411
barre de progression (JS) 293
 d'un traitement 275
brèves Client Web (RSS 2.0) du JDN
 Développeur 489
complétion avancée de bibliothèques
 Ruby 382
complétion simple de bibliothèques Ruby 376
consulter le Standblog (Atom 1.0) 499
décoration automatique de labels 106
deux listes à trier avec échange 367
didacticiel technique 575
Draggable avancé 349
Effect.Highlight 329
Effect.Opacity 325
Effect.Parallel 335
Effect.Parallel avec rappel 337
Effect.Scale 327
formulaire sémantique et accessible 566
jeu de photos Flickr 456
liste unique à trier 364

exemple (*suite*)

- nombre d'articles de blogs (DOM3
XPath) 287
- nombre d'articles de blogs
(GoogleAJAXSLT) 291
- prévisions meteo TWC 431
- saisie de commentaires 280
 - Prototype 312
 - script.aculo.us 340
- sauvegarde automatique 267
 - Prototype 304
 - suivi de marchés 296
- tableau de données à en-têtes groupés 573
- validation automatique de formulaires 111
- expression rationnelle 30, 165, 427

F

Firebug, extension Firefox 634

Firefox

- Firebug, extension 634
- roi des navigateurs de développeurs 634
- Web Developer Toolbar, extension 634

Flickr

- afficher une photo et ses informations 477
- obtenir les informations du jeu de photo 463
- obtenir une clef API 457
- récupérer les photos du jeu 471
- requête et réponse REST 458

Flotr 545

flux

- contenu HTML 488
- informations génériques 488

G

Google Chart 531

- alternatives 545

cartographies 543

couleurs 541

dégradé 542

encodage

- étendu 538

- simple 537

- textuel avec échelles 535

- textuel simple 534

enrobage JavaScript 546

gradient 542

paramètres obligatoires 531

personnalisation 534

plug-in rails 546

principe 531

tailles et limites de taille 534

types de graphique 539

URL de base 531

Google Earth 9

Google Maps 7, 520

- ajouter des contrôles 524

- ajouter des marqueurs 525

- alternatives à 530

- cartes non terrestres 530

- clé API 521

- créer une carte 522

- exigences techniques 520

- gérer les zooms 523

- modifier les icônes des marqueurs 528

Google Suggest 2

GreaseMonkey XXXVIII

H

hasFeature 80

HTML XXVIII

- HTML 4.01 XXIX, XXXVI

- HTML 5 XXIX

HTMLDocument, interface DOM 81

HTMLElement, interface DOM 82

I

IDL (Interface Description Language) 613

IETF XXXI, 608

in, opérateur JavaScript 35

inspecteur DOM 87

Internet Explorer Developer Toolbar 646

isLoaded, extension pour images 186

ISO/IEC 16262 21

J

JavaScript XXVIII, 19, 22

- ==, opérateur 35

- || pour des valeurs par défaut 51

- « fuites » par non-déclaration 24

accessibilité 103
arguments des fonctions 46
astuces 57
binding 47
conventions de nommage 60
équivalences booléennes 36
espaces de noms 54
eval 27
exceptions 36
fonctions globales 27
for...in 32
héritage de prototypes 41
in, opérateur 35
isNaN 27
JavaScript 1.6 21
JavaScript 1.7 XXXVI, 21
JavaScript 1.5 21
labélisation de boucles 31
lisibilité 59
mythes 20
New Function (éviter !) 53
objets anonymes 53
opérateurs 34
parseInt et parseFloat (mystères) 28
parseInt, préciser la base 28
portée 33
prototypes 41
rvalue à gauche de == 57
structures de contrôle 31
types de données 25
undefined 28
Unobtrusive JavaScript 55
variables déclarées ou non 23
with, opérateur 33

JScript 22

JSON (JavaScript Object Notation) 15, 166, 296

L

LiveScript 20

M

Mashups 519
MIME
type JavaScript 310

MSIE (Internet Explorer Developer Toolbar) 646

N

NamedNodeMap, interface DOM 78
Navigateur web (développer avec son navigateur) 631
Netvibes 4
Node, interface DOM 70
NodeList, interface DOM 78

O

objets anonymes (JavaScript) 53
obtenir une clef API
Amazon.fr 408
Flickr 457
The Weather Channel 430
ORB (Object Request Broker) 405

P

parseInt et parseFloat (mystères) 28
parseInt, préciser la base 28
plan d'actions 15
Platypus XXXVIII
POO 168
prefs.js (fichier) 21
Prototype 119
\$break 130
\$super 170
_each 151
AbstractObserver, objet 195
Ajax (petits secrets supplémentaires) 316
Ajax.activeRequestCount 316
Ajax.Autocompleter
 Ajax.XSLTCompleter 451
 updateChoices 451
Ajax.Base 306
Ajax.PeriodicalUpdater 314
Ajax.Request 304
 méthodes et options 306
Ajax.Response 309
Ajax.Updater 310
 différencier entre succès et échec 313
 options 311
Alias 123

Prototype (*suite*)
Array.clear 149
Array.clone 150
Array.compact 149
Array.concat 150
Array.first 148
Array.flatten 149
Array.from 147
Array.indexOf 148
Array.inspect 147
Array.intersect 149
Array.last 148
Array.lastIndexOf 148
Array.reduce 149
Array.reverse 149
Array.size 150
Array.toArray 150
Array.uniq 149
Array.without 149
Class.addMethods 169
Class.constructor 169
Class.create 168
Class.subclasses 168
Class.superclass 168
Classes et héritage 168
document.loaded (propriété personnalisée) 206
document.observe 198
dom (loaded) 206
élément étendu 122
Element, objet 171
Element, syntaxe constructeur 177
Element.absolutize 183
Element.addClass 178
Element.addMethods 185
Element.adjacent 180
Element.ancestors 179
Element.childElements 179
Element.cleanWhitespace 175
Element.clonePosition 184
Element.cumulativeOffset 183
Element.cumulativeScrollOffset 183
Element.descendantOf 179
Element.descendants 179

Element.down 180
Element.empty 175
Element.getDimensions 182
Element.getHeight 182
Element.getOffsetParent 183
Element.getOpacity 179
Element.getStyle 178
Element.getWidth 182
Element.hasAttribute 174
Element.hasClassName 178
Element.hide 174
Element.identify 187
Element.immediateDescendants 179
Element.insert 176
Element.inspect 186
Element.makeClipping 185
Element.makePositioned 182
Element.match 179
Element.Methods, module 171
Element.next 180
Element.nextSibling 179
Element.positionedOffset 183
Element.previous 180
Element.previousSibling 180
Element.readAttribute 173
Element.relativize 183
Element.remove 175
Element.removeClass 178
Element.replace 175
Element.scrollTo 185
Element.select 180
Element.setOpacity 179
Element.setStyle 178
Element.show 174
Element.siblings 179
Element.toggle 174
Element.toggleClassName 178
Element.undoClipping 185
Element.undoPositioned 183
Element.up 180
Element.update 175
Element.viewportOffset 183
Element.visible 174
Element.wrap 176

Element.writeAttribute 173
Enumerable, module 151
Enumerable.all 153
Enumerable.any 153
Enumerable.collect 156
Enumerable.detect 154
Enumerable.each 151
Enumerable.eachSlice 157
Enumerable.entries 159
Enumerable.filter 154
Enumerable.find 154
Enumerable.findAll 154
Enumerable.grep 155
Enumerable.include 153
Enumerable.inGroupsOf 157
Enumerable.inject 157
Enumerable.inspect 159
Enumerable.invoke 158
Enumerable.map 156
Enumerable.max 155
Enumerable.member 153
Enumerable.min 155
Enumerable.partition 154
Enumerable.pluck 156
Enumerable.reject 154
Enumerable.select 154
Enumerable.size 154
Enumerable.sortBy 158
Enumerable.toArray 159
Enumerable.zip 158
espaces de noms et modules 121
événements personnalisés 205
Event, objet 197
Event.element 200
Event.findElement 200
Event.fire 205
Event.inspect 204
Event.isLeftClick 203
Event.isMiddleClick 203
Event.isRightClick 203
Event.memo (propriété personnalisée) 205
Event.observe 198
Event.pageX (propriété garantie) 203
Event.pageY (propriété garantie) 203
Event.pointer 203
Event.pointerX 203
Event.pointerY 203
Event.preventDefault 202
Event.relatedTarget 201
Event.stop 202
Event.stopObserving 198
Event.stopped 202
Event.stopPropagation 202
exemple (sauvegarde automatique) 304
Extensions de Array 147
Extensions de String 140
Field, module/objet 191
Field.activate 192
Field.clear 191
Field.EventObserver, objet 207
Field.focus 191
Field.getValue 192
Field.Observer, objet 196
Field.present 192
Field.select 191
Field.serialize 192
Field.Serializers, objet technique 192
fonction \$ 125
fonction \$\$ 129
fonction \$A 126
fonction \$F 128
fonction \$H 127
fonction \$R 128
fonctions globales 124
Form, module/objet 193
Form.disable 193
Form.Element, module/objet 191
Form.enable 193
Form.EventObserver, objet 206
Form.findFirstElement 194
Form.focusFirstElement 194
Form.getElements 193
Form.getInputs 193
Form.Observer, objet 195
Form.request 317
Form.reset 193
Form.serialize 194
Form.serializeElements 194

Prototype (*suite*)

Function.argumentNames 135
Function.bind 133
Function.bindAsEventListener 134
Function.curry 136
Function.defer 136
Function.delay 136
Function.methodize 138
Function.wrap 137
gestion unifiée des événements 196
gestionnaires et binding 199
Hash, objet 159
Hash.clone 160
Hash.get 161
Hash.index 161
Hash.inspect 161
Hash.keys 160
Hash.merge 161
Hash.set 160
Hash.toObject 161
Hash.toQueryString 161
Hash.toTemplateReplacements 162
Hash.unset 161
Hash.update 161
Hash.values 160
héritage et la surcharge de méthode 170
initialize, rôle de constructeur 168
itérateurs 122
JSON 166
manipulation d'éléments 171
manipulation de formulaires 190
modules et objets génériques 151
Number.succ 140
Number.times 140
Number.toColorPart 139
Object, espace de noms 131
Object.clone 133
Object.extend 133
ObjectRange, objet 162
ObjectRange.include 162
PeriodicalExecuter, objet 162
PeriodicalExecuter.stop 163
POO 168

Prototype, objet global 207
Prototype.Browser 207
Prototype.BrowserFeatures 208
Prototype.emptyFunction 207
Prototype.K 207
Selector, objet 188
Selector.findChildElements 190
Selector.findElement 190
Selector.findElements 190
Selector.inspect 190
Selector.match 190
Selector.matchElements 190
Selector.toString 190
String.blank 147
String.camelize 142
String.capitalize 143
String.empty 147
String.endsWith 147
String.escapeHTML 142
String.evalJSON 167
String.evalScripts 144
String.extractScripts 144
String.gsub 143
String.include 147
String.inspect 145
String.interpolate 145
String.interpret 143
String.isJSON 167
String.parseQuery 145
String.scan 146
String.startsWith 147
String.strip 141
String.stripScripts 141
String.stripTags 141
String.sub 143
String.succ 146
String.times 146
String.toArray 145
String.toQueryParams 145
String.truncate 141
String.underscore 143
String.unescapeHTML 142
Template, objet 163
Template.evaluate 163

toElement(), protocole de conversion 175
toHTML(), protocole de conversion 175
toJSON 167
utilisation dans une page web 124
versions 120
vocabulaire et concepts 121

R

rafraîchissement strict 632
RATP, site d'itinéraires 2
RDF (Resource Description Framework) 486
REST (REpresentational State Transfer) 401,
 406
 exemple
 Amazon.fr 411
 jeu de photos Flickr 456
 prévisions meteo TWC 431
 serveur « proxy » 414
RFC (Request For Comments) 608
 1766 565
 2219 625
 2223 625
 2616 628
 2822 491
 3285 625
 4287 487
 822 491
RMI (Remote Method Invocation) 405
RSS (Really Simple Syndication) XXIX, 486
 0.9x 486
 1.0 486
 2.0 486
 exemple (brèves Client Web du JDN
 Développeur) 489
 channel 488
 items 488
Ruby 247
 installation 248
 premier petit serveur 252

S

Safari
 menu Debug 639
sauvegarde automatique 10

script.aculo.us 321
Ajax.AutoComplete 374
Autocomplete.Base 373
Builder 390
complétion
 automatique de texte 373
 incrémentale 383
Draggable 347
 options 347
Draggables.addObserver 352
Draggables.register 352
Draggables.removeObserver 352
Draggables.unregister 352
Droppables 352
Droppables.add 353
Droppables.remove 353
Effect.Appear 338
Effect.BlindDown 338
Effect.BlindUp 338
Effect.DropOut 338
Effect.Event 323
Effect.Fade 338
Effect.Fold 338
Effect.Grow 338
Effect.Highlight 323
 options 329
Effect.Morph 323
Effect.Move 323
Effect.Opacity 323
Effect.Parallel 323
Effect.Puff 339
Effect.Pulsate 339
Effect.Queue 344
Effect.Queues 344
Effect.Scale 323
 options 327
Effect.ScopedQueue 344
Effect.ScrollTo 339
Effect.Shake 339
Effect.Shrink 339
Effect.SlideDown 339
Effect.SlideUp 339
Effect.Squish 339
Effect.SwitchOff 339

script.aculo.us (*suite*)
Effect.toggle 339
Effect.Transform 323
Effect.Tween 323
effets
fonctions de rappel 325
invocation et options 323
option queue 344
options communes 324
effets combinés 338
effets noyau 323
effets visuels 322
exemple
complétion avancée de bibliothèques
Ruby 382
complétion simple de bibliothèques
Ruby 376
deux listes à trier avec échange 367
Draggable avancé 349
Effect.Highlight 329
Effect.Opacity 325
Effect.Parallel 335
Effect.Parallel avec rappel 337
Effect.Scale 327
liste unique à trier 364
saisie de commentaires 340
files d'effets 342
glisser-déplacer 346
pour trier 362
In Place Editing 390
Slider 390
Sortable 363
Sortable.create 363
Sortable.destroy 372
Sortable.serialize 372
tri par glisser-déplacer 362
Seamonkey XXXV
serveur HTTP minimaliste 520
services web 401, 405
SOAP (Simple Object Access Protocol) 405
spécification
descriptions de propriétés CSS 615
descriptions de propriétés et méthodes
DOM 616

DTD 607, 618
IDL 613
principaux formats 607
recommandation W3C 607, 608
RFC 608, 625
savoir lire une spécification 605
schéma XML 607, 622
spécifications phares 629
standards du Web XXVII
avantages d'utiliser XXXIII
inconvénients d'ignorer XXXII

T
Technorati XXXVIII
Template, objet
motif personnalisé 165
Text, interface DOM 77
The Weather Channel 430
obtenir une clef API 430
requête et réponse REST 433
Thomas Fuchs 322
transformation XSLT 418
TWC (The Weather Channel) 430
type MIME JavaScript 310
Typo XXXIX

U
Unobtrusive JavaScript 55

W
W3C XXIX
W3DTF 502, 514
WaSP XXXI
Web 2.0 XXXVII
définition XXXVII
Web Applications 1.0 XXIX
Web Developer Toolbar, extension Firefox 634
Web Forms 2.0 XXX
Web Standards Project XXXI
WHAT WG XXIX
with, opérateur JavaScript 33
WS-* 405
WSDL (Web Service Description Language) 405
WSH (Windows Scripting Host) 22

X**XHTML XXVIII**

- attributs incontournables 565
- avantages insoupçonnés 552
- balises
 - de formulaires et d'interaction 563
 - de liaison 560
 - de métadonnées 561
 - dépréciées 564
 - par catégorie 557
 - présentationnelles 561
 - sémantiques 558
 - structurelles 557
- différences entre 1.0 et 1.1 556
- DTD 555
- exemple
 - didacticiel technique 575
 - formulaire sémantique et accessible 566
 - tableau de données à en-têtes groupés 573
- prologue (XML) 555
- règles syntaxiques 553
- versions 556
- XHTML 1.1 XXXVI
- XHTML 2.0 556

XML XXVIII

- XMLHttpRequest XXVIII, 245
- créer le requêteur 255

envoi de requête 258

historique 254

méthode

- open 258

- send 258

- setRequestHeader 258

paramétrage 257

propriété

- onreadystatechange 259

- readyState 259

- responseText 259

- responseXML 260

- status 259

traitement de réponse 259

XPath 404

XSL (eXtensible Stylesheet Language) 404

XSL-FO (XSL-Formatting Objects) 404

XSLT (XSL-Transform) 404

- Ajax.XSLTCompleter 451

- chargement centralisé parallèle 461

- transformation 418

XUL (XML - based User Interface) 21

XULRunner 21

Y

Yahoo! Maps 530

Bien développer pour le Web 2.0



Adieu, soupes de balises et combinaisons de Javascript propriétaires qui polluaient le Web 1.0 ! Place à des applications accessibles et ergonomiques, des scripts portables et du balisage sémantique : créer des interfaces bluffantes et interactives à la Web 2.0 (Gmail, Google Maps, Flickr, Netvibes...) est l'occasion d'instaurer de bonnes pratiques de développement — pour travailler mieux, plus vite, et dans le respect des normes.

Une bible des meilleures pratiques de développement Web 2.0

Christophe Porteneuve livre dans cet ouvrage plus de dix années d'expérience en développement et en qualité web. Il rappelle les fondamentaux techniques du Web 2.0 (XHTML, CSS, JavaScript, DOM...), décrit l'usage des frameworks de développement dédiés Prototype et script.aculo.us dans leur version la plus récente, et explore le cœur d'Ajax, XMLHttpRequest, ainsi que la question des contenus et services externes (services web, API REST et flux de syndication RSS et Atom). Outre une réflexion sur l'accessibilité et l'ergonomie, il explique comment conjuguer toutes ces technologies dans le cadre d'une méthodologie de développement cohérente et qualitative.

Cette deuxième édition augmentée, entièrement mise à jour pour tenir compte des récentes évolutions des standards, illustre la technologie de la réutilisation d'API JavaScript tierces («mashups») et explore en détail le débogage JavaScript côté client de vos applications web.

Au sommaire

Web 2.0 et standards du Web • Mythes et rumeurs • Intérêts stratégiques • Rappels JavaScript • Opérateurs méconnus • Exceptions • Héritage de prototypes • Binding • Idiomes intéressants • Fonctions et objets anonymes • Simuler des espaces de noms • Bonnes pratiques d'écriture • Manipulations dynamiques avec le DOM • Niveaux DOM • Ordre des nœuds • Scripter au bon moment • Pas d'extension propriétaire • Gestion propre des événements • Accommoder MSIE • Capture et bouclonnement • Besoins fréquents : décoration automatique de labels, validation automatique de formulaires • Prototype : simplicité, portabilité et élégance • Accès au DOM • Tableaux et tableaux associatifs • Itérateurs • String enrichi • Des tableaux surpuissants : conversions, extractions, transformations • Éléments étendus • Manipulations des styles et classes • Modification de contenu • Parcours de hiérarchies • Positionnement • Manipulation de formulaires • Événements • Déboguer du JavaScript • Démarrer sur Firefox avec Firebug • MS Script Debugger • Visual Web Developer Express • Outils IE8 • Outils Safari 3 • Opera Dragonfly • Ajax, ou l'art de chuchoter • XMLHttpRequest • Anatomie d'une conversation Ajax • Préparer un échange asynchrone • ActiveX versus objet natif JavaScript • Créer l'objet requêteur, décrire et envoyer la requête, recevoir et traiter la réponse • Types de réponse : XHTML, XML, JS, JSON... XPath • GoogleAJAXSLT • Ajax avec Prototype • Ajax.Request • Ajax.Response • Ajax.Updater • Différencier la mise à jour entre succès et échec • Ajax.PeriodicalUpdater • Petits secrets supplémentaires • Script.aculo.us pour l'ergonomie • Effets visuels • Invocation • Options communes • Fonctions de rappel • Files d'effets • Glisser-déplacer avec Draggable et Droppables • Tri de listes • Complétion automatique de texte • Avoir du recul sur Ajax • Ajax et l'accessibilité • Services web et REST • Contraintes de sécurité • API REST • Exemple d'Amazon.fr • De XML à XHTML : la transformation XSLT • API météo • API Flickr • Flux RSS et Atom • Récupérer et afficher des flux • Feuille XSLT • Traiter des quantités massives de HTML encodé • Mashups • 100% côté client • Google Maps • Google Chart • Annexes • XHTML sémantique • CSS 2.1 • Le «plus» de l'expert : savoir lire une spécification • Les recommandations du W3C • Les RFC de l'IETF • Développer avec son navigateur web • Problèmes de cache • WebDeveloper • IE Developer Toolbar • Outils IE8 • Outils Safari 3 • Outils Opera • Les autres frameworks JavaScript • jQuery • Dojo • YUI • MooTools.

À qui s'adresse cet ouvrage ?

- Aux développeurs web qui doivent actualiser leurs connaissances et découvrir les technologies du Web 2.0 ;
- À ceux qui souhaitent explorer en profondeur les bibliothèques Prototype et script.aculo.us ;
- À tous ceux qui souhaitent acquérir une méthodologie cohérente de développement web, combinant technologies de pointe, qualité et accessibilité.

C. Porteneuve

Directeur technique à Ciblo et président 2008 de l'association Paris-Web, Christophe Porteneuve conçoit des pages web depuis bientôt 15 ans.

Après avoir participé au premier portail JSP en Europe (Freesbee), il dirigea la filière Systèmes d'Information et Génie Logiciel de l'INSIA pendant 5 ans, où il enseigne alors, entre autres, XHTML 1 Strict sémantique, CSS 2 et Ajax. Il est aujourd'hui contributeur à Ruby On Rails, script.aculo.us et Prototype. Son blog remonte à 2002, et il participe chaque année à diverses conférences de premier plan telles que Paris-Web, Paris On Rails et The Ajax Experience.

Code éditeur : G12391
ISBN : 978-2-12-12391-3



Conception : Nord Campo

www.editions-eyrolles.com

Groupe Eyrolles | Diffusion Geodif | Distribution Sodis

45 €