1

Introduction à Ajax

Inventé début 2005 par Jesse J. Garrett, le terme Ajax (Asynchronous JavaScript And XML) désigne un ensemble de technologies existant depuis plusieurs années, dont une utilisation ingénieuse rend possibles des fonctionnalités Web novatrices et utiles, qui rencontrent un succès grandissant depuis l'apparition d'applications telles que Google Suggest, Google Maps, writely, etc.

Grâce à Ajax, il est possible de bâtir des applications Web au comportement très proche de celui des applications Windows ou MacOS natives. L'avantage essentiel d'Ajax réside dans une plus grande réactivité de l'interface par rapport au Web classique.

Ce chapitre vise à situer Ajax dans le monde du développement Web. Il servira en outre de base aux chapitres ultérieurs.

Il commence par expliquer ce qu'est Ajax et les avantages qu'il apporte aux utilisateurs, puis répertorie les cas typiques où il leur est utile. Plusieurs des exemples cités sont mis en œuvre dans le reste de l'ouvrage. Le chapitre est illustré par un exemple simple et démonstratif, dont le code est détaillé.

Qu'est-ce qu'Ajax ?

Nous allons tout d'abord considérer l'aspect fonctionnel d'Ajax, afin de montrer ce qu'il apporte par rapport au Web classique.

Nous détaillerons ensuite les technologies mises en œuvre, puisque Ajax n'est pas en lui-même une technologie, mais un ensemble de technologies existantes, combinées de façon nouvelle.

Mise à jour d'une partie de la page

Dans une application Web classique, lorsque l'utilisateur clique sur un lien ou valide un formulaire, le navigateur envoie une requête au serveur HTTP, qui lui retourne en réponse une nouvelle page, qui remplace purement et simplement la page courante, comme l'illustre la figure 1.1.

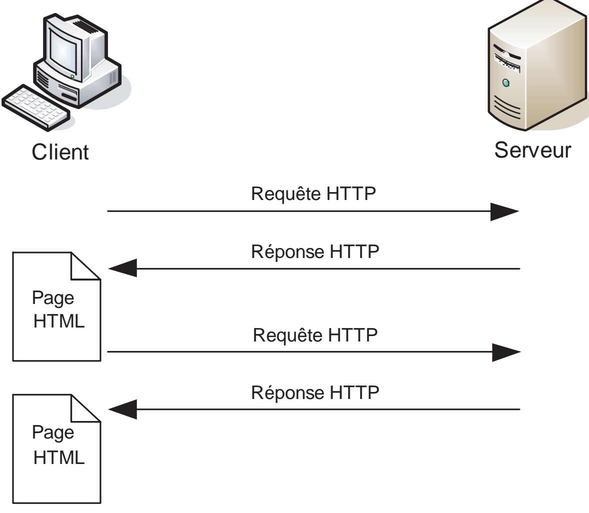


Figure 1.1
Communication client/serveur en Web classique

Par exemple, sur un site de commerce en ligne, l'utilisateur demande une page, dans laquelle il saisit ses critères de recherche de produits. Lorsqu'il valide le formulaire, une nouvelle page lui indique les résultats. Il peut alors, en cliquant sur le lien adéquat, ajouter tel produit à son panier, ce qui ramène une nouvelle page, par exemple la vue du panier, etc.

Dans la plupart des sites Web, les pages ont généralement des parties communes. Il s'agit notamment des liens vers les actions possibles sur le site, lesquels conservent le contexte et permettent à l'utilisateur de savoir où il en est et d'accéder rapidement aux informations ou aux actions possibles. Ces liens forment une sorte de menu, situé le plus souvent en haut ou à gauche des pages. Par exemple, pour un site de commerce, le panier de l'utilisateur est visible sur toutes les pages ou est accessible depuis un lien (souvent sous la forme d'une icône représentant un Caddie).

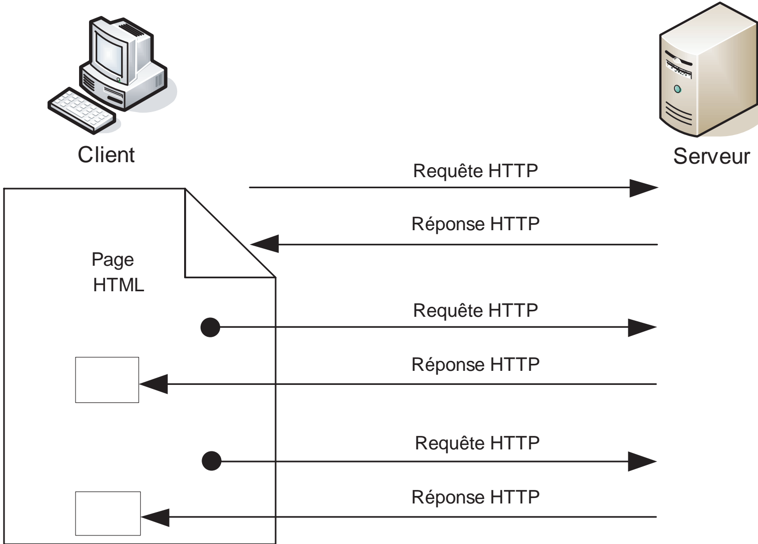
Avec le Web classique, ces parties communes sont envoyées avec chaque réponse HTTP. Lorsque le volume de données des éléments communs est faible par rapport à celle des éléments propres à la page, c'est sans grande conséquence. Dans le cas contraire, ce fonctionnement consomme inutilement une partie de la bande passante et ralentit l'application. Par exemple, si l'utilisateur modifie la quantité d'un article dans son panier, seuls deux ou trois petites portions de la page devraient être mises à jour : la quantité (un champ de saisie) et le nouveau montant de la ligne et du total. Le rapport est dans ce cas totalement disproportionné.

Avec Ajax, il est possible de ne mettre à jour qu'une partie de la page, comme l'illustre la figure 1.2. Les requêtes HTTP sont envoyées par une instruction JavaScript en réaction à une action de l'utilisateur. La réponse HTTP est également récupérée par JavaScript, qui peut dés lors mettre à jour la page courante, grâce à DOM et aux CSS, qui constituent ce qu'on appelle le HTML dynamique.

Plusieurs requêtes peuvent ainsi être émises depuis une même page, laquelle se met à jour partiellement à chaque réponse. Le cas extrême est constitué par une application réduite à une seule page, toutes les requêtes étant émises en Ajax. Nous verrons au chapitre 7 dans quels cas ce choix se montre judicieux.

Dans l'exemple précédent, lorsque l'utilisateur change la quantité d'un produit dans son panier, une requête HTTP est envoyée par JavaScript. À réception de la réponse, seules les trois zones concernées sont mises à jour. Le volume transitant sur le réseau est ainsi réduit (drastiquement dans cet exemple), de même que le travail demandé au serveur, qui n'a plus à reconstruire toute la page. La communication peut dés lors être plus rapide.

Figure 1.2
Communication client-serveur en Ajax



Communication asynchrone avec le serveur

La deuxième caractéristique d'Ajax est que la communication avec le serveur *via* JavaScript peut être *asynchrone*. La requête est envoyée au serveur sans attendre la réponse, le traitement à effectuer à la réception de celle-ci étant spécifié auparavant. JavaScript se charge d'exécuter ce traitement quand la réponse arrive. L'utilisateur peut de la sorte continuer à interagir avec l'application, sans être bloqué par l'attente de la réponse, contrairement au Web classique. Cette caractéristique est aussi importante que la mise à jour partielle des pages.

Pour reprendre notre exemple, si nous réalisons en Ajax l'ajout d'un produit au panier, l'utilisateur peut ajouter un premier article, puis un second, sans devoir attendre que le premier soit effectivement ajouté. Si la mise à jour des quantités achetés est également réalisée en Ajax, l'utilisateur peut, sur la même page, mettre à jour cette quantité et continuer à ajouter des articles (ou à en enlever). Nous mettrons d'ailleurs en œuvre ce cas au chapitre 7.

Les requêtes peuvent ainsi se chevaucher dans le temps, comme l'illustre la figure 1.3. Les applications gagnent ainsi en rapidité et réactivité.

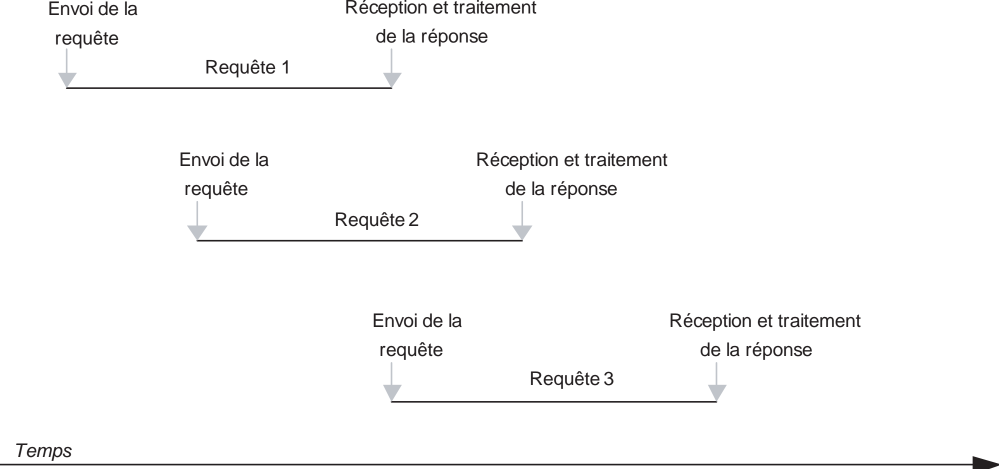


Figure 1.3
Requêtes Ajax asynchrones s'exécutant en parallèle

Le parallélisme des requêtes HTTP en Ajax est très utile dans la plupart des cas. Il exige cependant parfois de prendre certaines précautions, comme nous le verrons au chapitre 4.

Par exemple, si l'utilisateur valide le panier, il faut lui interdire de le modifier tant que la validation n'est pas terminée ou abandonnée, afin d'éviter de possibles incohérences. De même, cette validation doit être impossible tant que des requêtes modifiant ce panier sont en cours. Avant d'utiliser des requêtes asynchrones, il convient donc de vérifier qu'elles n'induisent aucun problème fonctionnel.

Techniques mises en œuvre

La communication avec le serveur repose sur l'objet JavaScript `XMLHttpRequest`, qui permet d'émettre une requête, de l'annuler (c'est parfois utile) et de spécifier le traitement à effectuer à la réception de sa réponse.

Le chapitre 4 examine en détail les problématiques relatives à cet objet. Disponible depuis 1998 dans Microsoft Internet Explorer, il l'est désormais dans tous les navigateurs récents.

Le traitement à effectuer lors de la réception de la réponse implique généralement de mettre à jour le contenu, la structure ou l'apparence d'une ou de plusieurs parties de la page. C'est précisément en quoi consiste le HTML dynamique.

Apparu dés 1996, le HTML dynamique a longtemps souffert d'incompatibilités des navigateurs, aujourd'hui limitées, et de l'aspect gadget de ce qui lui était souvent demandé (animations sans utilité fonctionnelle, rollovers, etc.). Le HTML dynamique repose sur DOM (Document Object Model), ou modèle objet du document, et les CSS (Cascading Style Sheets), ou feuilles de style en cascade, deux normes du Web Consortium aujourd'hui assez bien supportées par les navigateurs. Le HTML dynamique est présenté en détail au chapitre 2.

Avec Ajax, beaucoup de code peut être déporté du serveur vers le client. Le code JavaScript devient donc plus complexe, ce qui rend nécessaire de bien le structurer. C'est pourquoi les applications Ajax induisent une programmation objet en JavaScript, ainsi que l'utilisation d'aspects avancés de ce langage, lesquels sont couverts au chapitre 3.

Les réponses Ajax en HTTP peuvent être au format HTML, comme en Web classique, mais avec cette différence qu'il s'agit de fragments HTML, et non de pages entière, seule la portion de page à remplacer étant renvoyée.

Dans certains cas, cela se révèle tout à fait adapté, mais il peut être préférable dans d'autres cas de renvoyer la réponse sous forme structurée, de façon à pouvoir l'exploiter de plusieurs manières côté client. C'est là qu'intervient XML, qui préfixe l'objet `XMLHttpRequest`, ainsi que les techniques qui l'accompagnent, notamment XSLT. C'est aussi là qu'intervient JSON, un format d'échange adapté aux données, qui l'est parfois plus encore que XML. Le chapitre 5 se penche en détail sur ces deux formats.

Ajax repose ainsi sur tous ces piliers : DOM et CSS, JavaScript, `XMLHttpRequest`, XML et XSLT, ainsi que JSON.

Exemples typiques où Ajax est utile

Ajax est bien adapté à certains besoins mais pas à d'autres. Sans attendre d'avoir une idée approfondie de ses tenants et aboutissants, nous dressons dans cette section une liste des situations où il se révèle utile, voire précieux.

Plusieurs des exemples qui suivent seront mis en œuvre dans la suite de l'ouvrage.

Validation et mise à jour d'une partie de la page

Reprenons notre exemple de boutique en ligne.

L'utilisateur consulte un catalogue, ajoute des articles à son panier puis valide celui-ci et paie la commande générée. Lors de la validation du panier, si l'utilisateur ne s'est pas encore identifié, l'application lui demande de saisir son identifiant et son mot de passe ou de s'enregistrer si ce n'est déjà fait.

Prenons le cas d'un utilisateur déjà enregistré. Il est dirigé vers une page d'identification, puis, après validation, vers son panier. En Web classique, le serveur doit construire et renvoyer deux fois la page de validation du panier. Avec Ajax, nous pouvons rester sur cette page, y afficher le formulaire d'identification, et, lorsque l'utilisateur valide sa saisie, lancer en sous-main une requête au serveur, récupérer le résultat et l'afficher. C'est alors que l'utilisateur peut valider son panier.

Les figures 1.4 et 1.5 illustrent cet usage. Notons à la figure 1.5 l'icône indiquant que la requête est en cours de traitement.

Figure 1.4

Validation d'un formulaire sans changer de page

Figure 1.5

L'utilisateur a validé le formulaire, qui attend la réponse

L'avantage apporté par Ajax est ici double : l'application va plus vite, puisque nous réduisons le nombre de requêtes et la taille de la réponse (un simple OK, ou un message « Utilisateur inconnu »), et le déroulement est plus fluide pour l'utilisateur, qui peut même continuer sa saisie et changer, par exemple, la quantité à commander pour un article.

D'autres avantages d'Ajax se retrouvent dans d'autres parties du site. L'identification est généralement un prérequis pour des opérations telles que consulter ses commandes passées, spécifier des préférences (Ma recherche, Mon profil). Si la requête d'identification et sa réponse sont réduites au minimum, nous pouvons légitimement espérer un allégement non seulement d'une opération, mais de l'ensemble de celles qui sont concernées par l'identification.

Ce cas de figure se présente dans toutes les situations où une partie d'un formulaire peut être validée ou utilisée indépendamment du reste, notamment les suivantes :

- Quand l'utilisateur enregistre un client, pour vérifier l'existence de son code postal et proposer la ou les communes correspondantes. Le plus souvent, il y n'y en a une seule, mais, en France, il peut y en avoir jusqu'à 46 pour un même code postal.
- Dans une application d'assurance, pour déclarer un sinistre sur un véhicule. L'utilisateur indique l'identifiant du véhicule, puis, pendant qu'il saisit les informations concernant le sinistre, l'application interroge le serveur et ramène les données correspondant au véhicule.

Évidemment, si la requête au serveur est asynchrone, c'est-à-dire si elle ne bloque pas l'utilisateur, il faut lui montrer par un petit signe, comme le GIF animé de Mozilla pour le chargement, que la donnée est en train d'être récupérée. Ce *feedback* est très important.

Cet exemple est traité en détail à la fin de ce chapitre.

Aide à la saisie, notamment suggestion de saisie

Google Suggest a popularisé l'usage de l'aide à la saisie : l'utilisateur commence à saisir une expression, puis, à chaque caractère saisi, l'application interroge le serveur de façon asynchrone, récupère les 10 expressions les plus demandées commençant par l'expression saisie et les affiche sous forme de liste. L'utilisateur peut dés lors sélectionner l'une d'elles, qui se place dans le champ de saisie.

La figure 1.6 illustre un exemple de suggestion de saisie où l'utilisateur a entré « ajax ».

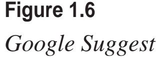


Figure 1.6
Google Suggest

Les serveurs de Google sont si rapides que les réponses aux requêtes sont quasiment instantanées. Toutes les applications ne disposent cependant pas d'une telle puissance. Il faut en tenir compte quand nous implémentons cette technique, en ne lançant des requêtes que tous les *n* caractères saisis, ou lorsque la saisie comprend déjà *n* caractéres.

Si nous reprenons l'exemple de la saisie d'un code postal, nous pouvons ne déclencher la requête destinée à récupérer les communes correspondantes que lorsque la saisie a atteint 5 caractères et annuler cette requête dés que l'utilisateur efface un caractére.

Nous pouvons même dans certains cas éviter tous les allers-retours vers le serveur, à l'image du site de la RATP. Pour fournir l'itinéraire d'une station de métro à une autre, cette application suggère les stations dont le nom commence par la saisie de l'utilisateur, et ce à chaque caractère entré. C'est aussi instantané que pratique. Il suffit, lors du chargement de la page, d'envoyer au navigateur un fichier contenant le nom de toutes les stations. Le poids de ce fichier n'étant que de 14 Ko, c'est une idée particulièrement judicieuse.

La question clé à déterminer pour savoir s'il faut envoyer les requêtes au serveur ou au client réside dans le volume des données dans lesquelles il faut rechercher. Pour les stations, il est si petit qu'il peut être envoyé intégralement sur le client. Pour les communes, ou des noms de clients dans une application d'assurance, il existe des dizaines ou des centaines de milliers d'enregistrements, si bien que des requêtes au serveur sont indispensables.

Cet usage d'Ajax si avantageux pour les utilisateurs exige en contrepartie une vérification attentive des performances ainsi que de sa pertinence dans chaque cas. Faire des suggestions quand l'utilisateur saisit un numéro de contrat n'apporte pas nécessairement grand-chose, alors que, pour le nom d'un produit ou d'un client, cela offre une sorte de visibilité de l'ensemble des valeurs, qui peut guider l'utilisateur.

La suggestion de saisie est traitée en détail aux chapitres 3 (en version locale) et 4 (en version communiquant avec le serveur).

Lecture de flux RSS

Les flux RSS sont une composante majeure de ce que l'on appelle le Web 2.0. Comme une page HTML, un flux RSS est la réponse à une requête faite à un serveur Web, dont la spécificité est de transmettre des nouvelles (agences de presse, journaux en ligne, blogs, nouvelles d'un site) ayant une structure prédéfinie. Cette structure contient essentiellement une liste d'articles portant un titre, un résumé, une date de mise en ligne et l'adresse de l'article complet. Qui dit information structurée, dit XML, si bien qu'un flux RSS est un document XML.

Avec Ajax, il est possible de récupérer de tels flux dans une page Web, de les afficher en HTML selon la forme souhaitée et de les faire se mettre à jour à intervalles réguliers.

Le site *www.netvibes.com* illustré à la figure 1.7 donne un exemple d'un tel usage. Nous y voyons plusieurs boîtes, une par flux : à gauche, un flux provenant de techno-science.net

et, à côté, un flux provenant du blog de netvibes, l'élément de droite étant simplement une zone de contenu statique.

Le site netvibes ajoute à cette fonctionnalité la possibilité de déplacer les boîtes sur la page par glisser-déposer, ce qui améliore encore son ergonomie. L'intégration d'une forte proportion de HTML dynamique est une constante des applications Ajax.

Une lecture de flux RSS proche de celle de *netvibes.com* est détaillée au chapitre 5.

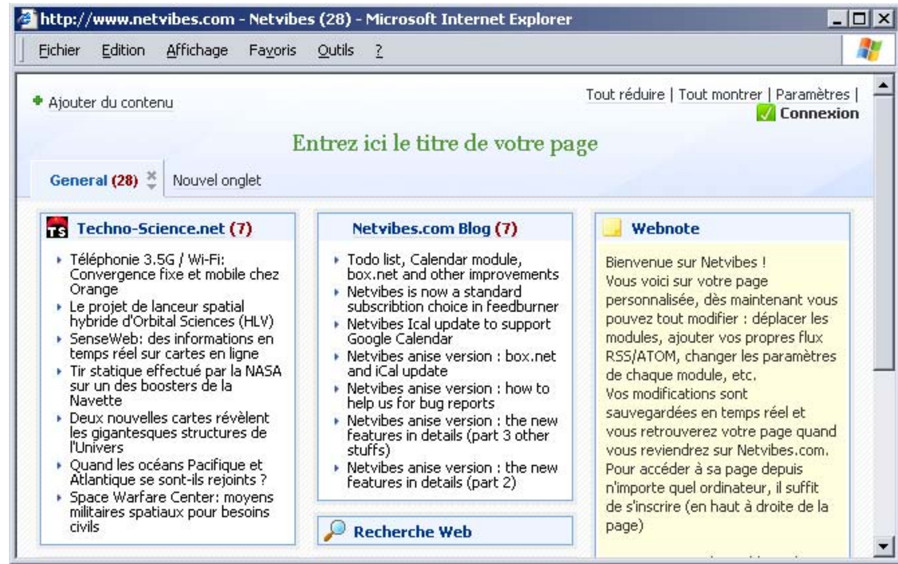


Figure 1.7
Affichage de flux RSS par le site netvibes

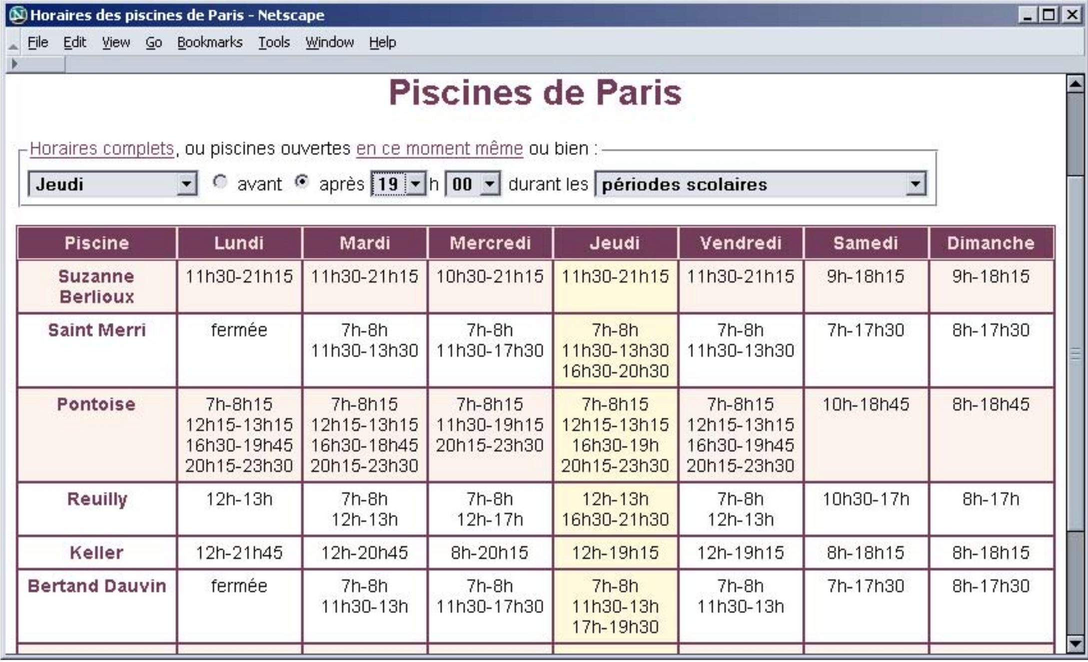
Tri, filtrage et réorganisation de données côté client

Il est très fréquent d'afficher à l'utilisateur les résultats d'une recherche sous forme de tableau, en lui permettant de trier les résultats par un clic sur une colonne, comme dans l'Explorateur de Windows, les logiciels de messagerie et bien d'autres applications. Il est aussi parfois utile de lui permettre de filtrer les résultats suivant tel ou tel critére.

Il arrive assez fréquemment que ces résultats ne dépassent pas la centaine ou un volume de l'ordre de 50 Ko, par exemple quand les données correspondent à une sous-catégorie telle que salariés d'un département dans une application de ressources humaines, produits d'un rayon dans une gestion de stock, etc.

Comme les données sont sur le client, ce serait du gâchis que de lancer chaque fois une requête au serveur. Une excellente façon de l'éviter est de transmettre les données en XML et de les afficher au moyen de transformations XSLT effectuées côté client. Les tris et filtrages étant instantanés, c'est là l'un des emplois les plus puissants d'Ajax et qui illustre non le A (asynchronous) de l'acronyme mais le X (XML).

La figure 1.8 illustre une page affichant des éléments d’une liste en fonction de critères de recherche.



Piscine	Lundi	Mardi	Mercredi	Jeudi	Vendredi	Samedi	Dimanche
Suzanne Berlioux	11h30-21h15	11h30-21h15	10h30-21h15	11h30-21h15	11h30-21h15	9h-18h15	9h-18h15
Saint Merri	fermée	7h-8h 11h30-13h30	7h-8h 11h30-17h30	7h-8h 11h30-13h30 16h30-20h30	7h-8h 11h30-13h30	7h-17h30	8h-17h30
Pontoise	7h-8h15 12h15-13h15 16h30-19h45 20h15-23h30	7h-8h15 12h15-13h15 16h30-18h45 20h15-23h30	7h-8h15 11h30-19h15 20h15-23h30	7h-8h15 12h15-13h15 16h30-19h 20h15-23h30	7h-8h15 12h15-13h15 16h30-19h45 20h15-23h30	10h-18h45	8h-18h45
Reuilly	12h-13h	7h-8h 12h-13h	7h-8h 12h-17h	12h-13h 16h30-21h30	7h-8h 12h-13h	10h30-17h	8h-17h
Keller	12h-21h45	12h-20h45	8h-20h15	12h-19h15	12h-19h15	8h-18h15	8h-18h15
Bertand Dauvin	fermée	7h-8h 11h30-13h	7h-8h 11h30-17h30	7h-8h 11h30-13h 17h-19h30	7h-8h 11h30-13h	7h-17h30	8h-17h30

Figure 1.8
Filtrage de données sur le poste client

Dés que l’utilisateur modifie un critère, le tableau change instantanément : seuls les établissements ouverts aux heures et jours indiqués sont affichés, et la colonne correspondant au jour courant (jeudi dans l’exemple) est mise en exergue (son fond change de couleur).

Nous traitons en détail au chapitre 5 un exemple équivalent, où les données sont des flux RSS.

Édition WYSIWYG de documents

Il s’agit ici de sauvegarder des documents plutôt que de soumettre des formulaires. Il peut s’agir de documents HTML ou de toutes sortes de documents structurés régulièrement manipulés par les entreprises, comme les notes de frais. L’utilisateur peut cliquer sur un bouton pour sauvegarder son document, tout en continuant à l’éditer. Pour le sauvegarder, l’application lance une requête asynchrone, dont le corps contient le document. L’édition fait pour sa part un recours massif au HTML dynamique.

Il faut prévoir un bouton Editer/Terminer l'édition, afin de verrouiller/déverrouiller le document en écriture au niveau serveur. Cela se révèle très pratique pour des documents assez riches et partagés, pour lesquels un formulaire aurait une structure trop rigide. Wikipedia est fournit un bon exemple.

Plusieurs sites proposent aujourd hui des applications bureautiques en ligne fonctionnant de cette mani re gr ce Ajax : traitement de texte (par exemple, writely), tableur, gestionnaire d agenda, sans parler des messageries. La figure 1.9 illustre l diteur de texte fourni par le framework Ajax dojo toolkit.

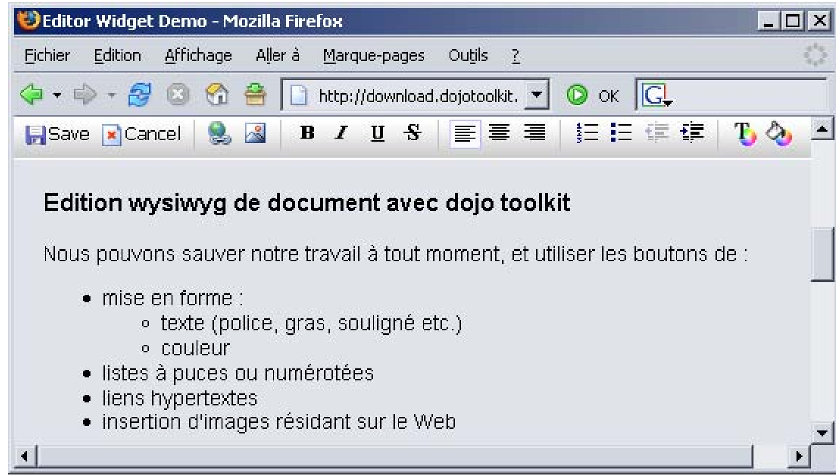


Figure 1.9
L'éditeur Ajax fourni par le framework dojo toolkit

Nous indiquons au chapitre 6 comment réaliser un tel éditeur.

Diaporamas et autres applications documentaires

Les applications documentaires se développent de plus en plus sur Internet. Il s'agit en particulier de disposer de différentes vues d'un même document : table des matières, fragment du document (page *n* d'un article découpé en *x* parties), intégralité du document sous forme imprimable. Nous pouvons, par exemple, avoir, à gauche d'une page, la table des matières et, à droite, la partie du document correspondant à l'entrée de la table des matières sur laquelle l'utilisateur a cliqué. Nous pouvons aussi faire défiler l'ensemble du document sous forme de diaporama.

J'utilise moi-même souvent cette technique pour des présentations. Le document mêle des éléments destinés à la présentation orale (titres et listes à puces) et du commentaire destiné à l'écrit. Il est possible d'imprimer tout le document sous forme de manuel et de l'afficher sous la forme d'un diaporama, avec en ce cas un affichage des caractères beaucoup plus gros.

La figure 1.10 illustre la création d'un diaporama sur le site *thumbstacks.com*.

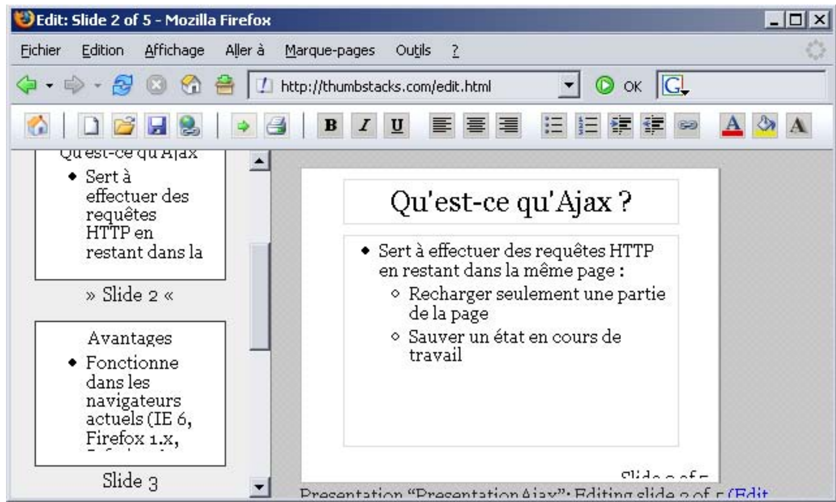


Figure 1.10
Édition d'un diaporama sur le site thumbstacks.com

Nous pouvons avoir des documents à structure libre (bureautique) ou souhaiter structurer plus fortement les documents grâce à XML en proposant à la consultation différentes vue résultant de transformations DOM ou XSLT côté client.

Pour l'utilisateur, l'avantage est là encore la rapidité de réaction que cela lui apporte. Le document volumineux est chargé en une fois, et le passage d'une vue à une autre est instantané. Pour l'éditeur du document, l'avantage est la garantie d'une cohérence entre les différentes vues, puisque toutes les vues sont obtenues à partir d'un unique document. Transposé dans le monde documentaire, nous retrouvons là le mécanisme des vues des bases de données relationnelles.

Débranchement dans un scénario

Reprenons l'exemple du panier dans une boutique en ligne.

Lorsque l'utilisateur commande sur le site pour la première fois, il doit s'enregistrer. Dans de nombreux sites, lorsque, sur la page du panier, il clique sur le bouton Valider le panier, il est redirigé vers une nouvelle page pour saisir son profil. Il remplit alors le formulaire et le soumet au serveur, après quoi l'application le connecte puis le ramène vers la page de commande (ou lui dit simplement « Vous êtes connecté », l'utilisateur devant encore cliquer sur Mon panier).

Cet usage est en quelque sorte une variante du premier (validation et mise à jour partielle).

Cela présente les deux inconvénients suivants :

- Performance. Le serveur doit traiter deux requêtes : l'affichage du formulaire d'enregistrement, d'une part, et sa validation et sa redirection vers la page du panier, d'autre part.
- Ergonomie. Du fait du changement de contexte (panier puis enregistrement), la fluidité du scénario est rompue.

Nous pouvons améliorer l'ergonomie d'un tel site en affichant le formulaire d'inscription dans une fenêtre pop-up ou dans un cadre interne. Dans les deux cas, nous réduisons le trafic puisque nous évitons le renvoi de la page du panier. Une autre façon de procéder consiste à déployer le formulaire d'enregistrement dans la fenêtre grâce à un clic sur un bouton ou un lien S'enregistrer et à ne soumettre au serveur que ce formulaire, par le biais d'une requête `XMLHttpRequest`.

Pour cela, il faut modifier dynamiquement le contenu ou l'affichage de la page et passer des requêtes partielles, c'est-à-dire sans redemander au serveur de reconstruire toute la page.

Un tel usage est étroitement lié à une règle de bonne conception des IHM : quand il y a plusieurs objets sur une page, nous réservons à chacun une zone sur cette page, et nous y plaçons les actions associées. Ici, nous avons les objets `catalogue`, `panier` et `utilisateur`. Nous pouvons disposer les trois objets sur la page, avec éventuellement une zone de travail additionnelle, et ne rafraîchir que ce qui est nécessaire.

Visualisation graphique avec SVG

Dans des applications d'aide à la décision (tableau de bord, analyse de données, prévision) ou de simulation, il est souvent très utile de présenter les données sous forme graphique : courbes, histogrammes, camemberts. Avant Ajax, nous étions réduits à générer ces graphiques côté serveur sous forme de fichiers GIF ou PNG et à les envoyer sur le client.

Nous pouvons aujourd'hui envisager de faire beaucoup mieux. SVG (Scalable Vector Graphics), une norme du W3C, est un type de documents XML qui décrit des graphiques vectoriels ainsi que leur animation. Comme il s'agit de XML, nous pouvons transformer les données en SVG au moyen de XSLT, en particulier sur le client et construire ainsi des applications dans lesquelles nous pouvons manipuler ces données côté client et en voir immédiatement le rendu graphique. Cela peut se faire directement dans Firefox à partir de la version 1.5 et dans Internet Explorer au moyen d'un plug-in comme le visualiseur SVG d'Adobe.

Rafraîchissement d'une branche d'une arborescence

Certaines applications présentent des informations sous forme arborescente. Un forum, par exemple, peut afficher une liste d'articles (news), dont chacun peut donner lieu à un fil de discussion, chaque réponse dans un fil pouvant à son tour donner lieu à un autre fil, etc. D'autres exemples de ce type sont fournis par les catalogues, avec leurs rubriques et sous-rubriques, les organigrammes d'entreprise *(voir figure 1.11)* ou encore les tables des matières des documentations.

Figure 1.11
Organigramme Ajax

Quand l'utilisateur cherche un élément de l'arborescence (dans notre exemple, Login Durand), nous l'affichons avec les nœuds ancêtres (service, direction) et voisins. Quand il veut détailler une branche de l'arborescence, nous récupérons par une requête asynchrone le détail de la branche et l'ajoutons simplement à l'arborescence, en conservant l'état déployé ou replié des autres branches.

L'utilisateur peut de la sorte parcourir tout l'arbre de façon flexible. Quand l'arbre est particulièrement imposant, c'est très utile. C'est ce que fait Microsoft pour la table des matières de son site de documentation technique *(msdn.microsoft.com).*

Chargement progressif de données volumineuses

Il arrive que les données à transmettre soient volumineuses. L'application Google Maps en est un bon exemple. Elle affiche une zone d'une carte, qu'il est possible de faire glisser de façon à afficher les zones voisines, et ce indéfiniment, et sur laquelle il est possible de faire des zooms avant et arrière. Cela fonctionne un peu comme Mappy, sauf que l'utilisateur peut faire glisser la carte sans aucune contrainte.

Grâce à Ajax, la carte à afficher est en fait découpée en petits morceaux. Quand l'utilisateur la déplace, le navigateur demande les morceaux voisins par des requêtes asynchrones. Le serveur répondant très vite, l'utilisateur a une impression de fluidité.

Exemple Ajax simple

Après avoir parcouru les différents usages d'Ajax dans les applications, nous allons nous pencher plus en détail sur un premier exemple simple, du point de vue tant fonctionnel que du code.

Nous reprenons le cas indiqué précédemment de la page affichant le panier d'une boutique en ligne et permettant de le valider. L'utilisateur dispose d'un formulaire pour cela,

mais il lui faut au préalable s'être identifié. La page sait s'il l'est en stockant cette information dans un champ caché ou dans une variable.

S'il n'est pas encore identifié, l'application lui propose de le faire à travers un autre formulaire dédié à cela. La page a alors l'aspect illustré à la figure 1.12 (l'emplacement du panier est simplement mentionné).

Figure 1.12
Formulaire d'identification

L'utilisateur remplit le petit formulaire puis le valide. L'application interroge alors le serveur sans demander une nouvelle page. En attendant la réponse (si le serveur est lent), l'utilisateur peut changer la quantité des articles qu'il a choisis. Il faut alors l'avertir visuellement que la requête est en cours de traitement, de même qu'il faudra l'avertir quand elle sera terminée.

Un moyen simple pour réaliser cela consiste à changer l'apparence du bouton de validation, en remplaçant le texte par une image animée suggérant le chargement, comme celle du navigateur Firefox. Il faut en outre donner à l'utilisateur la possibilité d'abandonner la requête, comme pour le Web classique, ce qui est l'objet du bouton Annuler, qui devient activé, ainsi que l'illustre la figure 1.13.

Figure 1.13
Attente de la réponse du serveur indiquée par une image animée

Si l'utilisateur annule la requête, le bouton S'identifier affiche à nouveau le texte initial, et le bouton Annuler est désactivé, comme l'illustre la figure 1.14.

Figure 1.14
Abandon de la requête

S'il laisse le traitement se poursuivre, une fois celui-ci terminé, la bouton Annuler est à nouveau désactivé. Si l'utilisateur a saisi les bons identifiant et mot de passe, l'application l'en avertit *(voir figure 1.15),* lui permettant de valider son panier en modifiant l'information correspondante stockée au niveau de la page. Dans le cas contraire, elle affiche un message d'erreur *(voir figure 1.16).*

Figure 1.15
Saisie validée par le serveur

Figure 1.16
Erreur indiquée par le serveur

Le code côté serveur

Bien que simple, cet exemple met déjà en œuvre plusieurs briques essentielles d'Ajax.

Nous allons examiner comment celles-ci s'imbriquent afin de bien faire comprendre les mécanismes mis en jeu dans Ajax. Nous étudions en détail chacune de ces briques au cours des chapitres suivants.

Nous écrivons le code serveur en PHP.

Nous avons les deux pages suivantes :

- **panier.php,** qui affiche le panier de l'utilisateur.
- **identifier.php,** qui identifie l'utilisateur, l'enregistre dans la session côté serveur s'il est reconnu et informe en retour s'il est connecté ou si les informations sont invalides. Ce service peut bien entendu être appelé depuis d'autres pages que **panier.php.**

Considérons rapidement le code de l'action **identifier.php.** Nous sommes dans du Web classique, avec une forme assez MVC (modèle, vue, contrôleur). La vue se charge de renvoyer le résultat à l'utilisateur. tandis que le modèle maintient l'état du serveur (session utilisateur, base de données). Pour la simplicité de l'exemple, il est réduit au minimum indispensable. Le contrôleur récupère les paramètres, interroge ou commande le modèle et demande à la vue de renvoyer la réponse.

Voici le code du contrôleur :

```
<?php
// Attendre 1 seconde pour simuler la realite : les serveurs
// sont souvent lents
/** Controleur */
sleep(1);
if (array_key_exists("login", $_POST)
  && array_key_exists("motPasse", $_POST)) {←❶
  // Les parametres ont bien ete transmis
  $user = get_user($_POST["login"], $_POST["motPasse"]);←❷
  if ($user) {
    connecter_user($user);←❸
  }
  repondre($user);←❹
}
else {
  print "usage : $_SERVER[PHP_SELF]?login=...&motPasse=...";←❺
}
```

En ❶, nous vérifions que nous avons bien les paramètres attendus. Si ce n'est pas le cas (ligne ❺), nous renvoyons le message indiquant l'usage de l'action. En ❷, nous récupérons l'utilisateur correspondant aux paramètres. S'il existe, nous le connectons (ligne ❸), et, dans tous les cas, nous renvoyons la réponse (ligne ❹).

Passons maintenant au code du modèle :

```
/** Modele */
```

```
function get_user($login, $motPasse) {
  // Le seul utilisateur valide sera Haddock/Archibald
  $user = null;
  if ($login == "Haddock" && $motPasse == "Archibald") {
    $user = array("id" => 1, "login" => "Haddock");
  }
  return $user;
}

function connecter_user($user) {
  // Enregistrer l'utilisateur dans la session web
}
```

Nous constatons que le code est vraiment réduit au minimum. La fonction `get_user` n'accepte qu'un utilisateur écrit en dur, sans aucun accès à la base de données. Si les paramètres lui correspondent, elle le renvoie, sinon elle renvoie `null`. De même, `connecter_user` a un corps vide.

La vue renvoie au client un simple message texte, sans aucun enrobage HTML ou XML (c'est lui qui est affiché sur le client, lequel le reprend purement et simplement) :

```
/** Vue */
function repondre($user) {
  if ($user) {
    print "Utilisateur '$user[login]' connect&eacute;";
  }
  else {
    print "Utilisateur inconnu ou mot de passe invalide";
  }
}
?>
```

Le code côté client

Examinons maintenant le code côté client de la page **panier.php.**

Au niveau HTML, nous avons un en-tête classique, pourvu d'une feuille de style réduite :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type"
      content="text/html; charset=iso-8859-1" />
    <title>Panier</title>
    <style type="text/css">
    button {
      width: 12ex;
      height: 2em;
      margin: 0ex 1ex 0ex 1ex;
    }
```

```
#panier {
  text-align: center;
  font-size:120%;
  background: #FAF0F5;
}
</style>
```

Suit du code JavaScript (examiné plus loin), puis le corps de la page, avec le panier (ici simplifié en un `div`) et un formulaire, dont l'action (ligne ❷) est un appel à la fonction JavaScript `identifier()` :

```
  <body onload="montrerInactivite()">←❶
    <p id="panier">Zone affichant le panier</p>
    <form name="identification"
      action="javascript:identifier()">←❷
      <table border="0">
        <tbody>
          <tr>
            <td>Identifiant</td>
            <td><input type="text" id="login"/></td>
          </tr>
          <tr>
            <td>Mot de passe</td>
            <td><input type="password" id="motPasse"/></td>
          </tr>
          <tr>
            <td colspan="2" style="text-align: center">
              <button type="submit">←❸
                <span id="identifierOff">S'identifier</span>
                <img id="identifierOn" src="loading.gif"
                  alt="Identification en cours ..."/>
              </button>
              <button type="button"
                id="boutonAbandonnerIdentifier"←❹
                onclick="abandonnerIdentifier()">Annuler</button>
            </td>
          </tr>
        </tbody>
      </table>
      <div id="message"></div>
    </form>
  </body>
</html>
```

Détaillons les trois fonctions JavaScript principales :

- `identifier()`, appelée en ❷, qui construit la requête HTTP et la lance. Le traitement est déclenché par le bouton S'identifier.
- `onIdentifier()`, qui correspond au traitement lorsque la réponse est complètement récupérée.

- `abandonnerIdentifier()`, qui abandonne la requête en cours d'exécution (équivalent du bouton Stop du navigateur). Cette fonction est déclenchée par le bouton Annuler (ligne ❹).

Voici le code de ces fonctions :

```
    <script type="text/javascript">
// La requete HTTP
var requete;

function identifier() {
  requete = getRequete();
  if (requete != null) {
    // Constituer le corps de la requete (la chaine de requete)
    var login = document.getElementById("login").value;
    var motPasse = document.getElementById("motPasse").value;
    var corps = "login=" + encodeURIComponent(login)
      + "&motPasse=" + encodeURIComponent(motPasse);
    try {
      // Ouvrir une connexion asynchrone
      requete.open("POST", "identifier.php", true);
      // Positionner une en-tete indispensable
      // quand les parametres sont passes par POST
      requete.setRequestHeader("Content-type",
        "application/x-www-form-urlencoded");
      // Traitement a effectuer quand l'etat de la requete changera
      requete.onreadystatechange = onIdentifier;
      // Lancer la requete
      requete.send(corps);
      // Montrer que la requete est en cours
      montrerActivite();
    }
    catch (exc) {
      montrerInactivite();
    }
  }
  else {
    setMessage("Impossible de se connecter au serveur");
  }
}

// Ce qui s'executera lorsque la reponse arrivera
function onIdentifier() {
  if (requete.readyState == 4 && requete.status == 200) {
    // Montrer que la requete est terminee
    montrerInactivite();
    // Afficher le message de reponse recu
    setMessage(requete.responseText);
  }
}

// Abandonner la requete
```

```
function abandonnerIdentifier() {
  if (requete != null) {
    requete.abort();
  }
  montrerInactivite();
  setMessage("Requ&ecirc;te abandonn&eacute;e");
}

// Recuperer la requete existante ou une nouvelle si elle vaut null
function getRequete() {
  var result = requete;
  if (result == null) {
    if (window.XMLHttpRequest) {
      // Navigateur compatible Mozilla
      result = new XMLHttpRequest();
    }
    else if (window.ActiveXObject) {
      // Internet Explorer sous Windows
      result = new ActiveXObject("Microsoft.XMLHTTP");
    }
  }
  return result;
}
```

Nous avons là les quatre fonctions utilitaires suivantes :

- `getRequete()`, qui renvoie la requête en cours et la crée si elle n'existe pas encore.
- `setMessage()`, qui affiche le message passé en paramètre.
- `montrerActivite()`, qui indique visuellement que la requête est en cours en activant le bouton Annuler et en affichant l'image de chargement dans le bouton de validation.
- `montrerInactivite()`, qui indique visuellement qu'il n'y a aucune requête en cours en désactivant le bouton Annuler et en affichant le texte « S'identifier » dans le bouton de validation.

Voici le code de ces fonctions :

```
// Mettre les boutons dans l'etat initial
function montrerInactivite() {
  document.getElementById("identifierOff").style.display = "inline";
  document.getElementById("identifierOn").style.display = "none";
  document.getElementById("boutonAbandonnerIdentifier").disabled = true;
}

// Montrer que la requete est en cours
function montrerActivite() {
  document.getElementById("identifierOff").style.display = "none";
  document.getElementById("identifierOn").style.display = "inline";
  document.getElementById("boutonAbandonnerIdentifier").disabled = false;
  setMessage("");
}
```

```
// Afficher un message
function setMessage(msg) {
  document.getElementById("message").innerHTML = msg;
}
```

La requête XMLHttpRequest

Tout au long de l'interaction avec l'utilisateur, nous avons besoin de manipuler la requête au serveur Web. Aussi la stockons-nous au niveau de la page dans la variable `requete`. Nous gérons une requête unique afin de pouvoir l'annuler quand l'utilisateur clique sur Annuler ou de l'annuler et la relancer avec la nouvelle saisie si l'utilisateur soumet à nouveau le formulaire (cela évite des requêtes et réponses inutiles).

La fonction `getRequete()` renvoie cette requête et la crée si elle n'existe pas déjà. S'il n'est pas possible de créer un tel objet, la fonction renvoie `null` (le navigateur n'est pas compatible Ajax).

Avec cette fonction, nous butons d'emblée sur les incompatibilités des navigateurs, qui est une des difficultés d'utilisation d'Ajax. Pour créer un objet de type `XMLHttpRequest`, nous sommes obligés d'avoir deux codes différents selon que nous sommes dans Internet Explorer ou dans un navigateur compatible Mozilla.

La fonction `identifier()` commence par récupérer un objet de type `XMLHttpRequest`. Dans le cas où il existe, elle constitue le corps de la requête en récupérant la saisie de l'utilisateur. Il faut cependant bien prendre garde à *encoder les paramètres saisis par l'utilisateur*, grâce à la fonction `encodeURIComponent()`, car c'est imposé par HTTP. Quand l'application utilise le mécanisme standard du navigateur, celui-ci encode automatiquement les paramètres. Ici, c'est au développeur de le faire.

Ensuite, la fonction ouvre la connexion en mode asynchrone (le troisième paramètre de `open()` vaut `true`), puis elle spécifie le traitement qu'il faudra exécuter lorsque la réponse sera reçue.

C'est ce que fait l'instruction suivante :

```
requete.onreadystatechange = onIdentifier;
```

qui signifie que, lorsque l'état de la requête (indiqué par l'attribut `readyState`) changera, le navigateur devra exécuter la fonction `onIdentifier()`. Il ne faut surtout pas mettre de parenthèses après `onIdentifier`, faute de quoi la ligne appellerait `onIdentifier()` et associerait le résultat obtenu, et non la fonction elle-même, au changement d'état.

Il ne reste plus qu'à indiquer à l'utilisateur que la requête est lancée, ce que nous faisons en appelant `montrerActivite()`.

Traitement asynchrone

À ce stade, la fonction `identifier()` a terminé son travail. Le fait que la requête soit asynchrone signifie qu'une tâche indépendante a été lancée. Le développeur en contrôle le déroulement dans la fonction `onIdentifier()`.

La requête passe par plusieurs états, numérotés de 0 à 4, dont la valeur est accessible dans l'attribut `readyState`. L'état qui nous intéresse est 4, le dernier, quand l'intégralité de la réponse a été récupérée. Nous vérifions aussi que le statut de la réponse HTTP vaut bien `200` (qui signifie OK). Dans ce cas, la fonction récupère le texte de la réponse et l'affiche simplement dans la zone de message.

HTML dynamique

Terminons cet examen du code avec les fonctions `montrerActivite()`, `montrerInactivite()` et `setMessage()`, qui utilisent deux méthodes du DOM HTML :

- `getElementById("unId")`, qui récupère l'élément de la page d'id `unId`.
- `innerHTML`, qui donne le contenu de l'élément sous la forme de son texte HTML. Cet attribut est en lecture-écriture (sauf exceptions).

Plusieurs éléments HTML sont justement pourvus d'un id :

- Le bouton Annuler.
- Le `div` (message) prévu pour afficher les messages.
- Les champs des formulaire `login` et `motPasse`.
- Les deux `span` (`identifierOff` et `identifierOn`) à l'intérieur du bouton de validation. Le premier contient le texte du bouton, et le second l'image indiquant que les données sont en cours de transfert depuis le serveur.

Au chargement de la page, le texte (`span identifierOff`) est visible, tandis que l'image (`span IndentifierOff`) est masquée (son style indique que la directive `display` vaut `none`). L'image doit être récupérée au chargement de la page, de façon à apparaître instantanément lorsque l'utilisateur valide le formulaire.

La fonction `montrerActivite()` montre l'image et cache le texte, tandis que la fonction `montrerInactivite()` fait l'inverse.

Conclusion

Le code JavaScript que nous venons de parcourir est représentatif de ce que fait Ajax. Nous y retrouvons la mise à jour partielle de la page, ainsi que le mécanisme d'appels asynchrones au serveur. Nous savons ainsi concrètement en quoi consiste du code Ajax.

Pour des raisons pédagogiques, nous avons conservé à ce code un style semblable à ce que nous trouvons habituellement dans les pages Web. Il faut toutefois savoir qu'avec Ajax, nous codons la plupart du temps en objet, car c'est notre intérêt. Pour cela, il faut bien connaître l'objet en JavaScript, qui est très particulier. Nous détaillons cette question au chapitre 3, et les chapitres 4 à 7 suivront ce style orienté objet.

Ajax peut être intégré à des applications existantes. Il faut pour cela ajouter du code JavaScript à des pages existantes, afin de gérer la communication *via* `XMLHttpRequest` ainsi

que la mise à jour de la page à la réception de la réponse. Cela peut nécessiter beaucoup de code JavaScript.

Il faut en outre modifier le code des réactions côté serveur, lesquelles, au lieu d'une page complète, doivent renvoyer un fragment HTML ou bien du XML, ou encore du JSON. Si notre application suit l'architecture MVC (modèle, vue, contrôleur), ces modifications sont cantonnées à la vue, qu'il s'agit alors de simplifier.

Nous nous retrouvons ainsi avec des vues plus complexes, comme **panier.php,** qui contiennent beaucoup de JavaScript, et d'autres réduites à leur plus simple expression, comme la fonction `repondre` de **identifier.php.**

L'une des parties les plus délicates concerne le HTML dynamique, auquel le Web 2.0 fait un appel massif. Pour cela, il faut bien connaître les techniques sous-jacentes et disposer de bibliothèques de composants graphiques. Il faut en outre être à l'aise avec les aspect objet et avancés de JavaScript.

Toutes les questions abordées dans ce chapitre seront donc approfondies tout au long des chapitres suivants. Nous commencerons par le HTML dynamique, sur lequel repose la modification des pages, puis nous passerons aux aspects avancés de JavaScript, de façon à être ensuite à l'aise pour traiter ce qui appartient en propre à Ajax, c'est-à-dire la communication avec le serveur *via* l'objet `XMLHttpRequest`, l'échange et la manipulation de données structurées et les bibliothèques JavaScript, parfois appelées abusivement frameworks Ajax.